OpenDoc Recipes

This chapter provides recipes for implementing most aspects of OpenDoc. In general, it provides greater detail than the rest of the *OpenDoc Programming Guide* .

This introduction explains which recipes part developers need to implement for different kinds of parts.

All part editors must implement:

- All Dynamic Binding recipes:
  - Binding
  - Installation of OpenDoc Software
  - Part Handler Registration
- All Part Persistency recipes:
  - Multiple Kind Support
  - Part Storage Model
  - Part Init and Externalizing
  - Part Info 'Ternalization
  - Display Frame 'Ternalization
  - Lazy Frame Internalization
- All Standards recipes:
  - Making Parent Method Calls
  - ODByteArray
  - Object Equality
  - Parameters and Memory in OpenDoc
  - Parameter Passing in Opendoc
  - Document Exchange
  - Part Properties
- All User Interface recipes:
  - Activation
  - Basic Event Handling
  - Dialogs
  - Menus
  - Pop-Up Menus
  - Opening and Closing Windows
  - Opening a Part into a Window
  - Window Events
  - Properties Notebook
  - Views
  - Help
  - Undo
  - Using Resources in OpenDoc
  - Facet Windows
  - Using Embedded PM Controls within a Facet
  - Implementing a Dispatch Module
  - Implementing a Focus Module
- Imaging and Layout recipes:
  - Adding a Display Frame
  - Adding and Removing Facets
  - Part Drawing
  - Printing
  - Determining Print Resolution
  - Scrolling
  - View Types and Presentations
  - Content Update Notification
  - Creating an EmbeddedFramesIterator
  - RequestEmbeddedFrame

Part editors which support data interchange (the clipboard, drag and drop, or linking) must implement:

- All Data Interchange recipes:
  - Clipboard Recipes
  - Data Interchange Basics
  - Drag and Drop

- Promises
- Drag and Drop - Promising Non-OpenDoc File
- Linking

Part editors which support embedding of parts must implement:

- Everything above plus...
  - Imaging and Layout recipes:
    -- Adjusting the Active Border
    -- Clipping Embedded Facets
    -- Embedding a Frame
    -- Making a Frame Visible
    -- Removing an Embedded Frame
    -- Frame Link Status
    -- Frame Groups and Sequencing

Other optional recipes or documents for informational purposes:

- Imaging and Layout recipes:
  - Creating and Using Offscreen Canvases
- Standards recipes:
  - Parameters and Memory in OpenDoc
- Storage recipes:
  - Storage Unit Manipulation
  - Persistent References
  - Reference Counting
  - Cloning Mechanism
  - Persistent Objects Removal
  - Alternate Container Suite Use
  - Bento Container Suite
- Shell Plug-In
- Public Utilities recipes:
  - Exception Handling
  - Temporary References and Objects

Standards

The standards recipes are listed as follows:

- Making Parent Method Calls
- ODByteArray
- Object Equality
- Parameters and Memory in OpenDoc
- Parameter Passing in Opendoc
- Document Exchange
- Part Properties

Making Parent Method Calls

According to *SOMobjects Developer Toolkit Users Guide* , page 4-56,

> "...the overriding method procedure can conveniently invoke the procedure that one or more of the parent classes uses to implement that method..." and

"The SOM-generated implementation header file defines the following macros for making parent-method calls from within an overriding method: <className>_parent_<parentClassName>_<methodName> (for each parent class of the class overriding the method), and

<className>_parents_<methodName>".

In OpenDoc, we recommend the use of the first form to make a parent-method call.

For example, if a part (called MyPart) needs to call its parent ODPart's InitPart method, it should use the following macro (defined in MyPart's implementation header file):

```
MyPart_parent_ODPart_InitPart(somSelf,ev,storageUnit,partWrapper)
```

Note: There are other macros that are generated by SOM for making parent-method calls. However, those short forms may be ambiguous if multiple inheritance is used. Also, some of these short forms exist only for backward compatibility and may not be supported in future versions of SOM.

ODByteArray

What Is an ODByteArray?
An ODByteArray is defined as a sequence of octets:

```
typedef struct
{
  unsigned long _maximum;
  unsigned long _length;
  octet *_buffer;
} _IDL_SEQUENCE_octet;
typedef _IDL_SEQUENCE_octet ODByteArray;
```

where:

_length is the number of bytes of relevant data.
_maximum is the number of bytes in the buffer.
_buffer is the pointer to the memory block containing the data.

Some rules about ODByteArray:

1. _length must be smaller or equal to _maximum.
2. kODNULL cannot be passed to any parameter which expects an ODByteArray pointer.
3. _buffer must be allocated by SOMMalloc if you are using a pointer to some type.

Why do we need ODByteArray?
The original OpenDoc function accepted and returned buffer pointers and their corresponding sizes. However, the function has been updated to use ODByteArray. This renders the OpenDoc function CORBA compliant and ready for distribution.

Usage example
The following examples creates ODByteArray on the stack. There is no restriction that ODByteArray is a stack variable. It can be allocated in the heap.

1. Setting a value on a storage unit with a pre-allocated buffer (ptr and size):

```
// Allocate the byte array on the stack
ODByteArray ba;

// Set up the byte array
ba._length = size;
ba._maximum = size;
ba._buffer = ptr;

// Get the value
```

```
su->SetValue(ev, &ba);

// ba will be deallocated automatically when the function exits.
// ptr needs to be explicitly deallocated by client.
```

2. Getting a value from a storage unit:

```
// Allocate the byte array on the stack. The fields are not pre-filled.
ODByteArray ba;

// Get the value
ODULong length = su->GetValue(ev, desiredLength, &ba);

// Use the buffer.
SomeFunction(ba._buffer);

// ba will be deallocated automatically when the function exits.
// ptr needs to be explicitly deallocated by client.
SOMFree(ba._buffer);
```

3. Getting a return value which is an ODByteArray:

```
// Byte array is allocated on the stack.
// None of the fields in the ODByteArray needs to be set prior to use.
ODByteArray ba = container->GetID(ev);

// Use the buffer in the ODByteArray.
  .
  .
  .
// Deallocate the buffer.
// Note that even though the client of the method does not allocate
// the memory, it is the client is responsibility to deallocate it.
SOMFree(ba._buffer);

// Byte Array is deallocated from the stack when the function exits.
```

Object Equality

This section describes how object references should be compared in OpenDoc and the motivation behind it.

IsEqualTo
In many places in the part code, object equality is determined by comparing pointers. This will fail when OpenDoc is used in a distributed environment. That is why ODObject has the method ODBoolean ODObject::IsEqualTo(in ODObject anotherObject). The purpose of this method is to determine whether two object refs refer to the same object. If so, kODTrue is returned. Otherwise, kODFalse is returned.

In addition, we have created a function in ODUtils called ODObjectsAreEqual. This function does a pointer comparison when it is legal to do so, thereby preventing the cost of calling ODObject::IsEqualTo.

In summary, it is required to use ODObjectsAreEqual or ODObject::IsEqualTo for all object pointer comparison.

Parameters and Memory in OpenDoc

This section describes the storage (or memory) responsibility of client arguments (or parameters). It first outlines the CORBA standard and

then it goes into specific details pertaining to OpenDoc.

CORBA Standard

In order to understand what are the storage (or memory) responsibility of client arguments, it is best to review the CORBA standard:

1. A client is responsible for providing storage for all arguments passed as in arguments.
2. A client is responsible for providing storage for all out arguments and return results except in the following cases:

	InOut	Out	Return Results
any	4	4	4
array	1	1	3
boolean	1	1	1
char	1	1	1
double	1	1	1
enum	1	1	1
float	1	1	1
long	1	1	1
object reference	2	2	2
octet	1	1	1
sequence	1	4	4
short	1	1	1
string	1	3	3
struct	1	1	1
ulong	1	1	1
union	1	1	1
ushort	1	1	1

Case 1
The client is responsible for providing storage and managing release of the storage. That is, the system allocates storage which must be free using ORBFree() for the following types and directions: string/out, string/return, sequence/out, sequence/return, array/return, any/inout, any/out, any/return. For inout strings and sequences, the out result is constrained by the size of the type on input.

Case 2
The client is responsible for providing the storage used to contain the object reference. When the reference is no longer needed, the client uses the release function to release the storage associated with the reference.

Case 3
The ORB provides the storage for these returned parameters and results. The client is responsible for releasing the storage using ORBFree.

Case 4
The client provides the storage for the structure which contains the description of the sequence or any and the client manages release of the storage and the descriptor. The ORB provides storage for the values returned and puts the pointers to this storage in the descriptor structures. The client is responsible for releasing this storage using ORBFree.

Refer to *The Common Object Request Broker: Architecture and Specification* page 98-99 for detail.

Implication to OpenDoc

If you are a client (that is, a caller) of an OpenDoc function:

	In	Out	InOut
Allocation	Stack	N/A	Stack
	SOMMalloc		SOMMalloc
Deallocation	SOMFree	SOMFree	SOMFree

If you are a callee (that is, methods of OpenDoc classes):

	In	Out	InOut
Allocation	N/A	SOMMalloc	SOMMalloc
Deallocation	N/A	N/A	N/A

Implication to Parts (that is, Client to OpenDoc Function):

	In	Out	InOut
Allocation	SOMMalloc	N/A	SOMMalloc
	Stack		Stack
Deallocation	SOMFree	SOMFree	SOMFree
	ORBFree	ORBFree	ORBFree

Other Rules

1. A callee must not modify in-parameters.
2. A callee must not store an in-parameter. You must duplicate the in-parameter.
3. Avoid ODObject/inout even though CORBA allows it.

   If you want to change the state or content of an ODObject, an in-parameter is sufficient. If you want to return an ODObject, an out-parameter should be used.

   ODObject/inout is strange because the incoming object may be deleted and a new object is created and returned. It is difficult to figure out whether the original object gets returned or not.
4. Do not reallocate memory for string/inout and sequence/inout.

   Since one cannot grow the buffer to more than the predetermined size, do not reallocate memory for these structures. Simply use the buffer provided.

Parameter Passing in Opendoc

This section outlines the cases where parameter passing can be problematic:

- Object Reference
- The string/in, string/out, or string/return parameter
- The sequence/in, sequence/out, or sequence/return parameter

---

Object Reference

Do not use ODObject/inout.

The string or ODISOString Parameter

1. In parameter:

   Example:

```
void StorageUnit::SetType(in ODValueType valueType);
```

   Client:
   Provides and releases the storage for the in parameter.

```
// First case using constant
const ODValueType kMyType = "MyValueType";
su->SetType(ev, kMyType);

// In this case, we do not need to explicitly dispose of the string
  .
  .
  .
// Second case using heap memory
ODULong strLength = ODISOStrLength(kMyType)+1;
ODPtr buffer = SOMMalloc(strLength);
memcpy(kMyType, buffer, strLength);
su->SetType(ev, buffer);

// In this case, we have to dispose of the string
ODDisposePtr(buffer);
```

   OpenDoc:
   Cannot store the incoming pointer. If it needs to store the string, it has to make a copy.

```
SOM_Scope void SOMLINK ODStorageUnitSetType(ODStorageUnit *somSelf,
                                            Environment *ev,
                                            ODType type)
{
  .
  .
  .
  ODISOStrCopy(type, _fMyType);

  // Do NOT dispose of memory
}
```

2. Out parameter:

   Example:

```
ODBoolean ODSession::GetType(in ODTypeToken token, out ODType type);
```

Client:
Releases the storage for the out parameter.

```
ODType type = kODNULL;

// Call OpenDoc
ODBoolean boolean = session->GetType(ev, token, &type);

// Use it
  .
  .
  .
// Client is responsible for disposing the out parameter
SOMFree(type);
```

OpenDoc:
OpenDoc provides the storage for the out parameter.

```
SOM_Scope ODBoolean SOMLINK ODBaseSessionGetType(ODBaseSession *somSelf,
                                                 Environment *ev,
                                                 ODTypeToken token,
                                                 ODType* returnType)
{
  .
  .
  .
  // Look up type using token
  ODType myType = ...

  // Allocate memory for the returned type.
  // Make sure that you make a copy of it before returning.
  *returnType = SOMMalloc(sizeOfMyType);

  // Copying the type
  ODISOStrCopy(myType, *returnType);

  return kODTrue;
}
```

3. Function return:

Example:

```
ODValueType ODStorageUnit::GetType();
```

Client:
Releases the storage for the function return.

```
// Call OpenDoc
ODValueType valueType = su->GetType(ev);

// Use it
  .
  .
  .
// Client is responsible for disposing the returned result
SOMFree(valueType);
```

OpenDoc:
OpenDoc provides the storage for the returned result.

```
SOM_Scope ODValueType SOMLINK ODStorageUnitGetType(ODStorageUnit *somSelf,
                                                   Environment *ev)
{
  .
  .
```

```
  .
  // Allocate memory for the returned type
  ODValueType returnValueType = SOMMalloc(sizeOfMyValueType);

  // Copying the value type
  ODISOStrCopy(myValueType, returnValueType);

  // Return the value type
  return returnValueType;
}
```

Sequence

1. In parameter:

   Example:

```
void ODStorageUnit::SetValue(in ODByteArray value);
```

   Client:
   Provides and releases the storage for the in parameter.

```
ODByteArray ba;
ba._length = length;
ba._maximum = maximum;
ba._buffer = buffer;
su->SetValue(ev, &ba);

// Client is also responsible for deallocating the memory for the buffer.
// That means the buffer is not consumed by the call.
```

   OpenDoc:
   Cannot store the incoming pointer. If it needs to store the string, it has to make a copy.

```
SOM_Scope void SOMLINK ODStorageUnitSetValue(ODStorageUnit *somSelf,
                                             Environment *ev,
                                             ODByteArray* value)
{
  .
  .
  .
  memcpy(value->_buffer, _fMyBuffer, value->_length);

  // Do NOT dispose of memory
}
```

2. Out parameter:

   Example:

```
ODBoolean ODValueNameSpace::GetEntry(in ODISOStr key,
                                     out ODByteArray value);
```

   Client:

   - Provides the storage for the structure which contains the description of the sequence.
   - Manages release of the storage and the descriptor.

```
// Create byte array
ODByteArray ba;

// Call OpenDoc
valueNameSpace->GetValue(ev, key, &ba);

// Use data returned by OpenDoc
  .
  .
  .
// Dispose the buffer returned by OpenDoc
SOMFree(ba._buffer);

// ODByteArray struct gets disposed when the function exits
```

OpenDoc:

- Provides storage for the values returned.
- Puts the pointers to this storage in the descriptor structures.

```
SOM_Scope ODBoolean SOMLINK ODValueNameSpaceGetEntry(ODValueNameSpace *somSelf,
                                                     Environment *ev,
                                                     ODISOStr key,
                                                     ODByteArray* value)
{
  // Look up the entry using key
  .
  .
  .
  // Determine actual length to be returned
  actualLength = ......

  // Allocate buffer for data to be returned
  buffer = SOMMalloc(actualLength);

  // Set the return byte array correctly
  value->_length = actualLength;
  value->_maximum = actualLength;
  value->_buffer = buffer;
  .
  .
  .
  return kODTrue;
}
```

3. Return result:

Example:

```
ODByteArray ODLinkSpec::GetPartData();
```

Client:

- Provides the storage for the structure which contains the description of the sequence.
- Manages release of the storage and the descriptor.

```
// Call OpenDoc
ODByteArray ba = linkSpec->GetPartData(ev);

// Use data returned by OpenDoc
  .
  .
  .
// Dispose the buffer returned by OpenDoc when done
SOMFree(ba._buffer);
```

OpenDoc:

- Provides storage for the values returned.
- Puts the pointers to this storage in the descriptor structures.

```
SOM_Scope ODByteArray SOMLINK ODLinkSpecGetPartData(ODStorageUnit *somSelf,
                                                    Environment *ev)
{
  .
  .
  .
  // Set up the local byte array
  ODByteArray ba;

  // Determine actual length to be returned
  actualLength = ......

  // Allocate buffer for data to be returned
  buffer = SOMMalloc(actualLength);

  // Set the return byte array correctly
  ba._length = actualLength;
  ba._maximum = actualLength;
  ba._buffer = buffer;
  .
  .
  .
  return ba;
}
```

Document Exchange

OpenDoc is intended to be a cross-platform architecture. Document Exchange is a general term referring to the ability for any platform to manipulate documents that are created on the same or other platform.

OpenDoc Meta-Data (persistent)

Data format

Date	Seconds since 1/1/70 (GMT)
Floating point	IEEE + size
Integer	1,2,4 bytes (signed and unsigned)
Boolean	1-Byte, 0 = FALSE
Fixed point	16.16
ISOString (non user-visible)	Null-terminated 7-bit ASCII
View type	ISOString
Presentation	ISOString
User-visible string	kODIntlTxt (ISO 10646,1993UCS standard)
Transform	3x3 Fixed, 16.16
Shape	Polygon

Alignment
Natural Alignment (that is, 8-bit Boundaries for bytes, 16-bit boundaries for 16-bit types and 32-bit boundaries for 32 bit types).

Coordinate system
ODPoint uses 16.16 Fixed and the default top-level coordinate system for an OpenDoc Document is one unit equals 1/72 of an inch.

Implications to Parts

OpenDoc meta-data and parts

In general, parts do not have to worry about the OpenDoc meta-data. The function hides most of the cross-platform details from the parts. For example, an ODShape associated with an ODFrame is stored persistently in little-endian format. When the frame internalizes the shape, it will do the necessary conversion so that the run-time ODShape object uses the endianess favored by the platform. Only when a part needs to access the OpenDoc meta-data stored in persistent storage directly does it have to worry about the persistent document exchange format.

Standardized data formats for parts
For OS/2 we have defined some standardized data formats (kinds). We are working with CI LABs to ensure that these become standard data formats for all OpenDoc platforms. Even if your part supports its own propriety format, it is useful to also support one of these formats, where appropriate. If several parts (of the same category) implement this mechanism, then there is some chance of data interchange between different machines and even different platforms.

The standardized data formats are:

kODKindPlainText	Unadorned ASCII text (codepage 850)
kODKindTextRTF10	RTF version 1.0
kODKindGraphicsCGM	CGM Graphics Metafile
kODKindImageGIF	GIF image
kODKindImageJPEG	JPEG image
kODKindVideoMPEG	MPEG video
kODKindAudioAU	AU Waveform Audio
kODKindMusicMIDI	MIDI

Cross-platform parts
Parts have complete control over their data format. Therefore, they do not have to follow the same standard as OpenDoc meta-data. However, one should try to make one's part as inspectable on persistent storage as possible.

Part Properties

Part Properties are items which affect a part's behavior or presentation, but which are not strictly held in the part's content. Typical examples are background color and default font, and it should be pointed out that these properties can be retained on a per-part or a per-frame basis.

Part frames in an OpenDoc document are maintained in a strict containment-based hierarchy, and a standard protocol has been established for allowing properties to be maintained and inherited through this hierarchy. This mechanism allows properties to be changed within a section of a document without having to make specific changes to each sub-component within that section. For example, a top-level page layout component can change the page background color and have that color change propagate in to a contained text part which contains a drawing part which contains a clock such that the backgrounds all remain in sync. It should also be noted that a given component is free to "de-couple" from the standard property and, for example, maintain an independent background color. In this case, it not only does not respect changes to this property from its container, it should explicitly prevent propagation of the property to its embeds (to maintain a strict property hierarchy).

Two things are required for this machinery to work properly: first, a standard set of properties must be defined, and second, components must implement the standard recipes for containing and embedded parts.

Standard Properties

Properties within OpenDoc are actually comprised of two aspects, the property name and the value type. The property name is a unique isostring which denotes the desired intent of the property, while the value type denotes the format the property is being held in. In fact, OpenDoc allows you to specify multiple value types if desired so you could pass (for example) a background color property with RGB and HSB values and a part inspecting the property would be free to use whichever was appropriate for it.

An initial set of agreed standard properties for use within OpenDoc for OS/2 is given here. For each, the #define name is shown for both the property and primary value-the actual isostrings can be found in STDPROPS.IDL in the OpenDoc toolkit:

Background color

property:	kODBackgroundColor
value:	kODRGB2

Foreground color

property:	kODForegroundColor
value:	kODRGB2

Default font

property:	kODFont
value:	kODFontNameSize (actually an ISOString with the same format as is used in OS/2 Presparams)

Background transparency

property:	kODBackgroundTransparency
value:	kODBoolean

An important point to note here is that while the first three of these properties can be thought of as properties of a container which embedded parts (frames) may or may not choose to respect, the background transparency should be thought of as a property of an embedded display frame which is controlled from the container side. Examples used in this recipe will pertain directly to the first three.

Non-Container Part Protocol

Non-container parts should maintain only those standard properties which are relevant to their function; it makes no sense for a non-container image part to concern itself with the default text font (unless its editing capabilities include creation of text on an image). Such Parts should retain the current values persistently, and handle changes to the values either through direct user interaction, such as selection from a menu or drag/drop from a system color palette, or through notification of changes from its container through the part's ContainingPartPropertiesUpdated method. A sample implementation for this method for non-container parts is:

```
SOM_SCOPE void SOMLINK MyPartContainingPartPropertiesUpdated( MyPart *somSelf,
                                    Environment *ev,
                                    ODFrame* frame,
                                    ODStorageUnit* propSU)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","ContainingPartPropertiesUpdated");

  RGBColor rgb;

  SOM_TRY

  // is the background color in there???
  if (ODSUExistsThenFocus( ev, propSU, kODBackgroundColor, kODRGB2) )
  {
    // read the value being passed in
    StorageUnitGetValue( propSU, ev, sizeof(rgb), &rgb);

    // it *should* be different, but check to be sure
    if (rgb != _fBGColor)
    {
      // Change our value.
      // This method should call SetChangedFromPrevious and invalidate the frame
      somSelf->SetBGColor(ev, frame, rgb );
    }
  }

  SOM_CATCH_ALL
  SOM_RETURN
}
```

In addition, when an embed is instantiated and its first display frame is added, it should ask its containing part for values for the standard properties using the AcquireContainingPartProperties method (of ODPart). It is conceivable that an embed not even retain the value and instead query the value whenever it was needed (such as when creating a thumbnail, or even when the Draw method was called), but calling up the hierarchy and building the set of standard properties each time should be expected to have relatively poor performance when compared to accessing instance data. An example of how to query your container's properties and extract the background color is:

```
SOM_Scope void  SOMLINK MyPartDisplayFrameAdded( MyPart *somSelf,
                                   Environment *ev, ODFrame* myFrame)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","DisplayFrameAdded");

  SOM_TRY

  ... other code

  // now lets get our container's std properties
  TempODFrame containingFrame = myFrame->AcquireContainingFrame(ev);
  TempODPart  containingPart = containingFrame->AcquirePart(ev);
  TempODStorageUnit suProps = containingPart->
                              AcquireContainingPartProperties(ev, myFrame);

  // Is the background color in there?
  if (ODSUExistsThenFocus( ev, propSU, kODBackgroundColor, kODRGB2) )
```

```
  {
    // read the value directly into our instance variable
    StorageUnitGetValue( propSU, ev, sizeof(_fBGColor), &_fBGColor);
  }

  SOM_CATCH_ALL
  SOM_RETURN
}
```

An additional point concerns whether or not an embed should respect its container's standard properties. Ideally, a part will not only retain its current value for a standard property of interest, but also a flag specifying whether this value is coupled to that of its container. This allows an embed to change its background color and know not to follow in response to its container sending a background color change notification. Also ideally, a user interface would be provided whereby the user could control this coupling. For cases where no direct mechanism is to be provided, an alternative protocol is proposed: the part should start out initially coupled with its container. If its background color is changed directly to something other than that of its container, it becomes de-coupled and stops responding to changes in this property from its container. Such a property can be re-coupled by changing the property in the embed to the same value that its container holds, as will be illustrated in the sample code for a container.

Container Protocol

Container parts have the greatest responsibility with regard to part properties. The are responsible for remembering and using the properties (as are embedded parts), propagating changes to embedded parts, and responding to queries from embedded parts. When a standard property is changed directly in a container, it is responsible for building a storage unit containing the modified property and passing the information to its embedded parts. This notification is performed by calling the part's ContainingPartPropertiesUpdated method on some or all of its embeds as appropriate. For a property like background color, the container should iterate over all of its embedded frames and make this call on the associated Part; for background transparency it is feasible to change the property for only those embeds that are selected.

When receiving a ContainingPartProperiesUpdated notification from a container's container, the secondary container has several responsibilities. First, it should inspect the passed-in storage unit for any properties that it is interested in for its own presentation that it is actively inheriting and adopt any changed values. Next, it should look for those properties that it is de-coupled on and determine whether to re-couple; this will typically be a simple compare with its current value. If there are properties of interest that the sub-container wishes to remain de-coupled from, it is responsible for taking them out of the storage unit using the Remove method. Finally, the sub-container should call ContainingPartPropertiesUpdated on its embeds to further pass along the notification. Thus, a container using background color might have an implementation like:

```
SOM_SCOPE void SOMLINK MyPartContainingPartPropertiesUpdated( MyPart *somSelf,
                                    Environment *ev,
                                    ODFrame* frame,
                                    ODStorageUnit* propSU)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","ContainingPartPropertiesUpdated");

  SOM_TRY

  // is the background color in there???
  if (ODSUExistsThenFocus( ev, propSU, kODBackgroundColor, kODRGB2) )
  {
    RGBColor rgb;
    StorageUnitGetValue( propSU, ev, sizeof(RGBColor), &rgb);

    // are we doing background color matching?  (its the default)
    if (_fMatchBGColor)
    {
      // change our value (do not propagate here, done below)
      // I've put a parameter on this method to signify whether to propagate
      // the color change in the SetBGColor call...
      somSelf->SetBGColor(ev, frame, rgb, kODFalse);

    // we are not following this change so we will NOT propagate this property
    }
    else
    {
      // If the new value matches our current value we re-couple.
      // Note that our embeds already know our color
      // so we don't need to pass it along.
      if (rgb == _fBGColor)
      {
        _fMatchBGColor = kODTrue;
      }

      // If this is the only property in the su, abort here
      // so as to NOT propogate it
```

```
      if (propSU->CountProperties(ev) == 1)
      {
        return;
      }
     // Otherwise remove this property and leave the rest
      else
      {
        propSU->Focus(ev, kODBackgroundColor, 0, 0, 0, kODPosUndefined);
        propSU->Remove(ev);
      }
    }
  }

  // now pass any remaining notification along to embeds
  ODFrame *subframe = (ODFrame*)_fEmbeddedFrames->First();
  while (subframe)
  {
     TempODPart embeddedPart = subframe->AcquirePart(ev);
     if (embeddedPart)
     {
        embeddedPart->ContainingPartPropertiesUpdated(ev, subframe, propSU);
     }
     subframe = (ODFrame*)_fEmbeddedFrames->After(subframe);
  }

  SOM_CATCH_ALL
  SOM_RETURN
}
```

It is important to note that a container should not call ContainingPartPropertiesUpdated on its embedded frames unless it is *changing* a properties value-it should, therefore, not call this for a newly-created embed. It is the responsibility of the embed to query its container if it is interested in using its standard properties. To respond to such a query, the container must implement AcquireContainingPartProperties. Such a request must be passed up the hierarchy to the document root part and each frame in the hierarchy must insure that the appropriate inheritance is maintained. An example of this is shown here:

```
SOM_Scope ODStorageUnit*  SOMLINK MyPartAcquireContainingPartProperties(
                                                MyPart *somSelf,
                                                Environment *ev,
                                                ODFrame* embedFrame)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","AcquireContainingPartProperties");

  SOM_TRY

  ODStorageUnit* propSU = kODNULL;
  TempODFrame myFrame = embedFrame->AcquireContainingFrame(ev);

  // sanity check
  if (!_fDisplayFrames->Contains(myFrame))
  {
    return kODNULL;
  }

  // if we are not the root part, pass the request up the chain
  if (!myFrame->IsRoot(ev))
  {
    TempODFrame containingFrame = myFrame->AcquireContainingFrame(ev);
    TempODPart  containingPart = containingFrame->AcquirePart(ev);

    propSU = containingPart->AcquireContainingPartProperties(ev, myFrame);
  }

  // If we are the root part OR our container didn't give
  // us back an SU, create one
  if (!propSU)
  {
    propSU = somSelf->GetStorageUnit(ev)->GetDraft(ev)->CreateStorageUnit(ev);
  }

  // default to leave the existing property alone
  ODBoolean wemadeit = FALSE;

  // if the std prop isn't there, create it and we write it
  if (!ODSUExistsThenFocus( ev, propSU, kODBackgroundColor, kODRGB2) )
    ODSUForceFocus( ev, propSU, kODBackgroundColor, kODRGB2 );
    wemadeit = kODTrue;
  }

  // If we created the prop OR we are de-coupled
```

```
  // from our container... write our color
  if (wemadeit || !_fMatchBGColor ) {
    StorageUnitSetValue( propSU, ev, sizeof(RGBColor), &_fBGColor);
  }

  SOM_CATCH_ALL
  SOM_RETURN

  // return the filled storage unit
  return propSU;
}
```

Additional Note for Root Containers

An additional note is that it can be beneficial for root containers to maintain values for the standard properties even if they do not use them directly. This would enable a user to make a change of any standard property at the root level and have it propagated throughout the document. For example, a PageLayout component that does not deal directly with text could still maintain the default font for drag and drop and even provide a menu item for changing the value, which could then be propagated to any embedded text-handling components. This would allow the user to change the default font for all text in a layout even though the root component does not handle text itself.

Notes on Background Transparency Handling

As mentioned above, background transparency should actually be thought of as a property of an embedded display frame that is controlled by its container. Because there is no AcquireEmbeddedPartCapabilities, it is not feasible for a 2-way dialog regarding transparency to occur so the following protocol is recommended.

By default containers and embeds should assume that the embed is opaque-that is, the value of kODBackgroundTransparency is kODFalse. As per the recipe for facet windows, to reduce flicker, the container can use the embed's UsedShape as a clip shape for rendering its own background; the embed has sole responsibility for rendering its area and should fill its background appropriately. If the user selects an embedded frame and changes its background rendering to transparent, the containing part should stop including the embed's UsedShape in its accumulated background clip shape. An example of processing for this is:

```
  // net embed shape for background clipping.  collect in WINDOW coords
  ODShape* contentClip = (ODShape*)facet->GetPartInfo(ev);
  // clear out any old clip shapes
  contentClip->Reset(ev);

  // utility shape
  TempODShape embedShape = facet->GetFrame(ev)->CreateShape(ev);

  // iterate over all embedded facets
  ODFacet* embFacet;
  ODFacetIterator* facets = facet->CreateFacetIterator(ev,
                                                       kODChildrenOnly,
                                                       kODFrontToBack);
  for (embFacet = facets->First(ev);
       facets->IsNotComplete(ev);
       embFacet = facets->Next(ev))
  {
    // get the embed's associated graphics object
    Proxy* p = somSelf->ProxyForFrame(ev, embFacet->GetFrame(ev));
    GtkEmbed* embedObj = (GtkEmbed*)_picture->Object(p->id);

    // if we got it and it is an embed that should be clipped off...
    if (embedObj && embedObj->IsEmbed() && !embedObj->IsTransparent()) {

      // get a copy of the embed's usedShape
      TempODShape tmpShape = embFacet->GetFrame(ev)->AcquireUsedShape(ev, kODNULL);
      embedShape->CopyFrom(ev, tmpShape);

      // convert to window coords
      xform = embFacet->AcquireWindowFrameTransform(ev, kODNULL);
      embedShape->Transform(ev, xform);
      ODReleaseObject(ev, xform);

      // add it to aggregate
```

```
      contentClip->Union(ev, embedShape);
    }
  }
  delete facets;
```

Whenever an area including a transparent embed is invalidated, the container will draw in the background first, before the embed is drawn. In addition to changing its clipping, the container should notify the embed that this property has changed. If the embed implements background transparency, it would then stop drawing its background in its Draw method, and only render its foreground (text or other content). This is implemented in the Draw method as follows:

```
SOM_Scope void  SOMLINK MyPartDraw( MyPart *somSelf,  Environment *ev,
                                    ODFacet* facet, ODShape* invalidShape)
{
    MyPartData *somThis = MyPartGetData(somSelf);
    MyPartMethodDebug("MyPart","Draw");

   SOM_TRY

    HPS hpsDraw;
    ODRect rect;
    TempODShape tempShape = facet->GetFrame(ev)->AcquireFrameShape(ev, kODNULL);
    tempShape->GetBoundingBox(ev, &rect);
    RECTL frameRect;
    rect.AsRECTL(frameRect);

    // Set up the drawing facet
    // Set up drawing environment
    CFocus f(ev, facet, invalidShape, &hpsDraw);

    // if we are not transparent, fill our background
    if (!_fTransparent)
    {
      GpiSetColor(hpsDraw, _fContentStruct.fColor);
      POINTL orig = {0, 0};
      GpiMove(hpsDraw, &orig);
      POINTL ptl = {frameRect.xRight, frameRect.yTop};
      GpiBox(hpsDraw, DRO_FILL, &ptl, 0, 0);
    }

    // draw our foreground stuff
    GpiSetLineType(hpsDraw, LINETYPE_DASHDOUBLEDOT);
    GpiSetColor(hpsDraw, CLR_BLACK);

    for (int y = 0; y < frameRect.yTop; y += YGRID) {
       ptl.y = y;
       ptl.x = 0;
       GpiMove(hpsDraw, &ptl);
       ptl.x = frameRect.xRight;
       ptl.y += frameRect.xRight;
       GpiLine(hpsDraw, &ptl);
    }

    for (int x = XGRID; x < frameRect.xRight; x += XGRID) {
       ptl.x = x;
       ptl.y = 0;
       GpiMove(hpsDraw, &ptl);
       ptl.x += frameRect.yTop;
       ptl.y = frameRect.yTop;
       GpiLine(hpsDraw, &ptl);
    }

   SOM_CATCH_ALL
   SOM_ENDTRY
}
```

If the embed does not implement background transparency, it will still render its background; the net result will be additional flickering whenever the embed is redrawn. It is clear that this is not an ideal mechanism for such communication-a more appropriate route would be to include transparency as a standard Container Suite OSA SetData property and use OSA for negotiating the cooperative handling. This PartProperties protocol is proposed as an intermediate mechanism for enabling more sophisticated cooperation until the OSA handling is standardized.

Storage

This section provides a roadmap for those who want to learn how to use the OpenDoc storage using the recipes. It is assumed that you have a basic knowledge of OpenDoc and its storage system (including container, document, draft and storage unit).

Storage recipes contains the following documents that discuss the basic storage concepts and usage:

- Storage Unit Manipulation
- Persistent References
- Reference Counting
- Cloning Mechanism
- Persistent Objects Removal
- Alternate Container Suite Use

The following recipe is specific only to the bento container suite:

- Bento Container Suite

Cloning Mechanism

Every OpenDoc draft contains a network of storage units. These storage units are connected to each other through persistent references. See the Persistent References recipe for more information. When an OpenDoc draft is opened, ODPersistentObjects and ODStorageUnit objects are instantiated using the storage units.

Copying an object is not as straight forward as the usual stream operations. For example, when one tries to copy a value, one needs to copy all the storage units that are being referred to by the value also. Otherwise, the data copied is not complete.

In order to help developers accomplish this task more easily, OpenDoc provides a cloning mechanism to enable deep-copy through a fairly simple function. This function has the following characteristics:

1. Clients have to go through source ODDraft to clone any object. ODDraft in turn calls the appropriate object's CloneInto method. Clients should never call the CloneInto method of another object directly.

2. Cloning is a transacted operation. Therefore, clients have to start the cloning operation by calling BeginClone and commit the cloning operation by calling EndClone. If for any reason, cloning cannot be completed, the client should call AbortClone to abort the transaction.

3. The cloning mechanism is biased toward running objects. Therefore, if there is a running object corresponding to a particular storage unit, the cloning mechanism will call the running object's CloneInto method instead of the storage unit's. This ensures that externalize does not need to be called before any data interchange.

4. The cloning mechanism respects a scope and the scope can only be a frame object or a storage unit belonging to a frame.

5. There are different kinds of cloning. They are mainly for data interchange purposes. See the Clipboard Recipes recipe for more information.

6. The ids returned from Clone and WeakClone cannot be used between BeginClone and EndClone. Therefore, if you want to use one of those ids for getting the object or making a reference, you will have to wait till after EndClone.

7. Some of ids returned may turn out to be invalid because the corresponding objects have not been strongly cloned. Refer to the appropriate recipes to determine what to do in specific situations.

Recipe for Clients

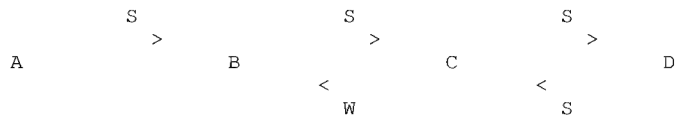
Clients need to use the following ODDraft function calls:

```
ODDraftKey BeginClone(in ODDraft destDraft,
                      in ODFrame destFrame,
                      in ODCloneKind kind);

void EndClone(in ODDraftKey key);

void AbortClone(in ODDraftKey key);
```

```
ODID Clone(in ODDraftKey key,
           in ODID fromObjectID,
           in ODID toObjectID,
           in ODID scope);

ODID WeakClone(in ODDraftKey key,
               in ODID objectID,
               in ODID toObjectID,
               in ODID scope);
```

BeginClone, EndClone and AbortClone are for setting up and terminating the cloning transaction. Clone and WeakClone are used to notify the draft that the specified object needs to be cloned.

1. Given the following network of objects, if the client wants to clone A, the client needs to do the following:




```
S = strong persistent reference to
W = weak persistent reference to
```

The object A has a strong persistent reference to B.
The object B has a strong persistent reference to C.
The object C has a strong persistent reference to D.
The object C has a weak persistent reference to B.
The object D has a strong persistent reference to C.

```
ODDraftKey key = draft->BeginClone(ev, destDraft, kODNULL, kODCloneCopy);
ODID newA = draft->Clone(ev, key, A->GetID(ev), 0, 0);
draft->EndClone(ev, key);
```

The result is that A, B, C, D are all copied.

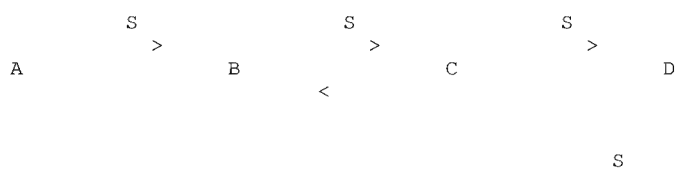
2. Given the same network of objects in step 1 if the client wants to clone C, the client needs to do the following:

```
ODDraftKey key = draft->BeginClone(ev, destDraft, kODNULL, kODCloneCopy);
ODID newB = draft->WeakClone(ev, key, B->GetID(ev), 0, 0);
ODID newD = draft->Clone(ev, key, D->GetID(ev), 0, 0);
draft->EndClone(ev, key);
```

The result is that C and D are copied. B is not copied because it is not strongly referenced by C or any objects transitively strongly referenced by C. The newB returned by WeakClone is an invalid ID. Therefore, when the client tries to get that object using newB, an error results. It is the responsibility of the client to handle this error condition.

Also, the ids can only be used to make storage unit references during cloning. The client should not try to get the persistent object or storage unit.

The following diagram shows what the copied network of storage units/persistent objects looks like:




```
S = strong persistent reference to
W = weak persistent reference to
```

The object A has a strong persistent reference to B.
The object B has a strong persistent reference to C.
The object C has a strong persistent reference to D.
The object D has a strong persistent reference to B.

3. If the network in step 2 looks like the following, the CloneResult will be different.

   In this case, the code fragment in step 2 copies C, D and B. Then, the returned newB is a valid ID referring to the new copy of B. The following diagram shows what the resulting network looks like:

Recipe for Parts

All persistent objects (including parts) have to override CloneInto in ODPersistentObject:

```
void CloneInto(in ODDraftKey key,
               in ODStorageUnit toSU,
               in ODFrame scope);
```

During the clone operation (that is, between BeginClone and EndClone), a persistent object's CloneInto method is called by the draft to do the actual data copying. The recipe is very similar to that for the clients except that the persistent object has to write out its intrinsic content in addition to cloning its referenced storage units/persistent objects.

The following is an example of what a part editor might do in response to CloneInto:

```
void MyPartCloneInto(MyPart* somSelf,
                     Environment* ev,
                     ODDraftKey key,
                     ODStorageUnit toSU,
                     OID scopeFrameID)
{
  // Since this method may be called multiple times during one cloning
  // transaction, the part should check to see whether it really needs
  // to write out any data.
  if (toSU->Exists(ev, kODPropContents, kMyContentKind, 0) == kODFalse)
  {
    // Write out intrinsic content
    toSU->AddProperty(ev, kODPropContents);
    toSU->AddValue(ev, kMyContentKind);
    toSU->SetValue(ev, myContentByteArray);

    // Write out referenced object
    ODStorageUnit* mySU= somSelf->GetStorageUnit(ev);
    ODDraft* myDraft = mySU->GetDraft(ev);
    ODID newReferencedObjectID =
    myDraft->Clone(ev, key, myReferencedObject, 0, scopeFrameID);

    // Write out referenced object id
    ODStorageUnitRef ref;
    toSU->GetStrongStorageUnitRef(ev, newReferencedObjectID, ref);
    toSU->SetOffset(ev, correctOffsetInMyContent);

    // Set up the byte array
    ODByteArray refByteArray;
    refByteArray._length = sizeof(ODStorageUnitRef);
    refByteArray._maximum = sizeof(ODStorageUnitRef);
    refByteArray._buffer = &ref;
    toSU->SetValue(ev, &refByteArray);
  }
}
```

Scoping for Cloning

If a part has two frames and the user only chooses to copy one frame, the part should be notified about it. Otherwise, the part may copy too

much data or in certain cases the wrong data. Let's consider the following situation. Part A contains part B (which has two display frames). In turn, part B contains part C and part D.

If the user selects some intrinsic content with B1 (as shown in the thin box) and selects Copy, part A should do the following:

```
// Begin cloning transaction
ODDraftKey key = draft->BeginClone(ev, destDraft, kODNULL, kODCloneCopy);

// Write out intrinsic content to storageUnit
  .
  .
  .
// Get reference to frame B1 using frame B1 as the scope
ODID newB1 = draft->Clone(ev, key, B1->GetID(ev), 0, B1->GetID(ev));

// Write out the reference to storageUnit also.
  .
  .
  .
// Finish cloning
draft->EndClone(ev, key);
```

When part B's CloneInto method is called, the scope parameter is the ID of frame B1. In this way, part B can choose to clone only the necessary data.

However, if the user selects both B1 and B2 (as shown below), Part A should do the following:

```
// Begin cloning transaction
ODDraftKey key = draft->BeginClone(ev, destDraft, kODNULL, kODCloneCopy);

// Write out intrinsic content to storageUnit
  .
  .
  .
// Get reference to frame B1 using frame B1 as the scope
ODID newB1 = draft->Clone(ev, key, B1->GetID(ev), 0, B1->GetID(ev));

// Get reference to frame B2 using frame B2 as the scope
ODID newB2 = draft->Clone(ev, key, B2->GetID(ev), 0, B2->GetID(ev));

// Write out the reference to storageUnit also
  .
  .
  .
// Finish cloning draft->EndClone(ev, key);
```

As a result, part B's CloneInto method is called twice (once with frame B1 as the scope and another time with frame B2 as the scope). Part B should be ready to handle this and write out the part content accordingly.

For more information on Data Interchange, see the Data Interchange recipes (especially the Clipboard Recipes recipe).

Destination Frame for Cloning

When the ODCloneKind argument to BeginClone is kODClonePaste or kODCloneDropMove, parts must supply as the ODFrame argument the frame that is performing the paste or receiving the drop. For other values of ODCloneKind, the frame may be kODNULL, as in the examples in this document. This parameter allows OpenDoc to determine if the destination is a valid location for content being moved; an invalid destination frame is one that would cause a display frame of a part to be embedded within another display frame of the same part. In this unusual situation, the subsequent call to EndClone raises an exception. When EndClone returns an error, parts must call AbortClone (which does not return errors).

Persistent Object Annotations

As discussed above, the cloning mechanism is biased toward run-time objects. Therefore, if there is anything in the storage unit that is not stored by the run-time persistent object itself, it is not going to be cloned. An example of this may be a utility like a spell checker. It may store a list of words in the part's storage unit.

In order to make sure that this information is not lost, the container suite copies any properties with the following prefix to the destination storage unit:

```
const ODPropertyName kODPropPreAnnotation=
                     "+//ISO 9070/ANSI::113722::US::CI LABS::OpenDoc:Annotation:";
```

The persistent object being cloned does not even need to know that these annotations exist.

Reference Counting

This section discusses the following:

- Why do we need ODRefCntObject?
- What is ODRefCntObject?
- What are the ODRefCntObjects in OpenDoc?
- How does it work?
- What are the implications on ODPersistentObjects?

Why Do we Need ODRefCntObject?

During an OpenDoc session, many objects are created. Because there is a very complex relationship among objects, it is difficult to determine when it is safe to delete an object.

Reference counting is a way to determine when these run-time objects can be deleted so that valuable memory space can be reclaimed.

What Is ODRefCntObject?

ODRefCntObject is an object with a reference count. A reference count must be 0 or a positive integer.

Here is the class definition of ODRefCntObject:

```
interface ODRefCntObject:ODObject
{
  void  InitRefCntObject();
  void  Acquire();
  void  Release();
  ODULong  GetRefCount();
};
```

What Are the ODRefCntObjects in OpenDoc?

These are the subclasses of ODRefCntObject class:

ODLink	(indiretly via ODPersistentObject)
ODLinkSource	(indiretly via ODPersistentObject)
ODPart	(indiretly via ODPersistentObject)
ODFrame	(indiretly via ODPersistentObject)
ODContainer	
ODDocument	
ODDraft	
ODStorageUnit	
ODWindow	
ODExtension	(and it's subclasses ODSemanticInterface, ODShellPlugIn, ODSettingsExtension)
ODShape	
ODTransform	

How Does it Work?

1. Every ODRefCntObject constructed has a reference count of 1.

2. Every ODRefCntObject is created or acquired from an object which keeps track of it (a.k.a. factory object).

ODRefCntObject	Factory Object
ODContainer	ODStorageSystem
ODDocument	ODContainer
ODDraft	ODDocument
ODExtension	ODObject (or subclass)
ODFrame	ODDraft
ODLink	ODDraft
ODLinkSource	ODDraft
ODPart	ODDraft
ODShape	ODFrame
ODStorageUnit	ODDraft
ODTransform	ODFrame
ODWindow	ODWindowState

For example, to get a ODFrame object, one has to call ODDraft's CreateFrame or ODDraft's AcquireFrame. The ODFrame object returned from ODDraft's CreateFrame has a reference count of 1 while the ODFrame object returned from ODDraft's AcquireFrame has a reference count of at least 1.

Let's look at a very simple example:

```
// frame1's RefCount is 1
ODFrame* frame1 = draft->CreateFrame(...);

// frame2 == frame1 and its RefCount is 2
ODFrame* frame2 = draft->AcquireFrame(...);

// frame2's RefCount is 1.
// Since frame1 == frame2, frame1's RefCount is also 1.
frame2->Release();

// The user should not use frame2 after this point!
frame1->Acquire(. . .);  // frame1's RefCount is 2
frame1->Release(. . .);  // frame1's RefCount is 1
frame1->Release(. . .);  // frame1's RefCount is 0

// The user should not use frame1 after this point!
frame1->Acquire(. . .); or  // ERROR!
frame1->Release(. . .);      // ERROR!
```

3. When the RefCount goes down to 0, it is the object's responsibility to tell the factory object about it.

   For example, ODFrame's Release() can be implemented like this:

```
void ODFrameRelease(ODFrame* somSelf, Environment* ev)
{
  parent_Release(somSelf, ev); // which calls ODRefCntObject's Release()
  if (somSelf->GetRefCount(ev) == 0)
  fDraft->ReleaseFrame(ev, somSelf); // Call the factory object
}
```

The factory object can choose to dispose of the object immediately or keep the object around until a purge is called upon itself.

In the OpenDoc implementation, ODDraft uses the latter strategy. Therefore, ODDraft does not delete the object as soon as ODDraft's ReleaseXXX is called. Instead, it puts this released object in a collection so that it can be retrieved and reused again (that is, when ODDraft's AcquireXXX is called for that object). But, if ODDraft's Purge() is called, all the released objects in the collection will be deleted.

The reason why the object is not deleted immediately is that the object may be reused in the near future. One good example is scrolling. Parts may choose to release objects that are scrolled out of view and re-get them when they come into view again. If we delete the objects as soon as ReleaseXXX is called, we will have to re-instantiate these objects again. Creating persistent objects is not a fast process because internalization requires accessing the secondary storage (namely, the disk).

4. Every time a reference to a ODRefCntObject (that is, ODRefCntObject pointer) is cached in some data structure, ODRefCntObj->Acquire() should be called by the code that manipulates this data structure. When the code is finished with the ODRefCntObj reference in the data structure, it should call ODRefCntObj->Release().

   For example, since ODFrame is keeping a reference to the containing frame and the part, it has to increment the refCounts on both of these objects.

```
void ODFrameInitFrame(ODFrame* somSelf,
                      Environment* ev,
                      ODStorageUnit* storageUnit,
                      ODFrame* containingFrame,
                      ODShape* frameShape,
                      ODPart* part,
                      ODTypeToken viewType,
                      ODTypeToken presentation,
                      ODULong frameGroup,
                      ODBoolean isRoot,
                      ODBoolean isOverlaid)
{
  . . .
  _fContainingFrame = containingFrame;
  if (_fContainingFrame != kODNULL)
      _fContainingFrame->Acquire(ev);
  _fPart = part;

  if (_fPart != kODNULL)
      _fPart->Acquire(ev);
  . . .
}
```

In ODFrame's ReleaseAll(), the frame has to release the containing frame and the part. Otherwise, the containing frame and the part always have a RefCount greater than 0 and they can never be disposed of.

```
void ODFrameReleaseAll(ODFrame* somSelf, Environment* ev)
{
  if (_fContainingFrame != kODNULL)
      _fContainingFrame->Release(ev);

  if (_fPart != kODNULL)
      _fPart->Release(ev);
}
```

Note that the containing frame and the part may have RefCounts greater than 0 even after ODFrame's ReleaseAll() is called. This is because other objects may be keeping a reference to the containing frame or the part.

What Are the Implications on ODPersistentObjects?

When the RefCount of a persistent object goes from 1 to 0, the part has to call its factory object (that is ODDraft). The code looks something

like this:

```
void MyPartRelease(MyPart* somSelf, Environment* ev)
{
  parent_Release(ev); // Which calls ODRefCntObject's Release(ev)

  if (somSelf->GetRefCount(ev) == 0)
      _fDraft->ReleasePart(ev, somSelf);
}
```

As mentioned above, ODDraft does not delete the released object immediately. Therefore, a part does not need to do any shutting down or deallocation when its RefCount goes down to 0. They are then done in ReleaseAll() and the destructor (that is, somUninit).

One should not consider a persistent object with a zero refcount a dead object because at any point the draft may increment its refcount and hand it to some other objects. Having a refcount of 0 simply means no one has a reference to the object.

However, if a part needs to function differently when there is no reference to it, it can use the RefCount to identify the situation and respond appropriately. For example, it may also choose to get rid of any structure or service that is not needed anymore. For example, a Communication part may choose to close its driver when there is no reference to it.

```
void CommPartRelease(CommPart* somSelf, Environment* ev)
{
  parent_Release(ev);
  if (somSelf->GetRefCount(ev) == 0)
  {
     CloseDriver();
     fDraft->ReleasePart(ev, somSelf);
  }
}
```

If the object does something other than calling its factory object when its RefCount goes down to 0, it needs to reinitialize itself when its RefCount is bumped from 0 to 1. This is being done in the object's Acquire() call. In general, the object should undo what it did in Release() when the RefCount went down to 0.

The following is an example of what the above-mentioned CommPart would do in response to its RefCount going from 0 to 1.

```
void CommPartAcquire(CommPart* somSelf, Environment* ev)
{
  parent_Acquire(somSelf, ev);
  if (somSelf->GetRefCount(ev) == 1)
  {
     OpenDriver();
  }
}
```

Persistent References

A persistent reference is used in a storage unit value to refer to another storage unit in the same document. This reference is persistent in the sense that it is preserved across sessions.

References are used in many places in OpenDoc:

1. The draft has a list of references to the root frames. When a draft is opened, the root parts can use the frames to construct their windows.
2. Frame has a persistent reference to its part and the part in turn has a persistent reference to its display and embedded frames. Persistent references to embedded frames are essential to part embedding.
3. Parts need persistent references for hierarchical storage.
    - The draft properties have a persistent reference to the frame's storage unit.
    - The frame's storage unit has a persistent reference to its part's storage unit.

- The part's storage unit has a persistent reference to its frame's storage unit.
- The part's storage unit also has a persistent reference to its auxiliary storage unit.

ODStorageUnitRef

This persistent reference in OpenDoc is typed ODStorageUnitRef. ODStorageUnitRef is a 32-bit value. One should never try to inspect or interpret a ODStorageUnitRef. ODStorageUnit and ODStorageUnitView provides a set of methods to manipulate ODStorageUnitRef.

```
void GetStrongStorageUnitRef(ODStorageUnit* embeddedSU, ODStorageUnitRef ref);
void GetWeakStorageUnitRef(ODStorageUnit* embeddedSU, ODStorageUnitRef ref);

ODBoolean IsStrongStorageUnitRef(ODStorageUnitRef ref);
ODBoolean IsWeakStorageUnitRef(ODStorageUnitRef ref);

ODStorageUnit* RemoveStorageUnitRef(ODStorageUnitRef ref);
ODStorageUnitID GetIDFromStorageUnitRef(ODStorageUnitRef ref);

ODBoolean IsValidStorageUnitRef(ODStorageUnitRef ref);
```

ODStorageUnitRef can only be created from a ODStorageUnit which is focused to a value or a ODStorageUnitView (is to be focused to a value). The scope of the ODStorageUnitRef is then limited to the value from which it is created. It is illegal and dangerous to use ODStorageUnitRef in a different value. If used, it will almost certainly not refer to the correct storage unit rendering the resulting behavior unpredictable.

One can create virtually unlimited number of References in a storage unit value.

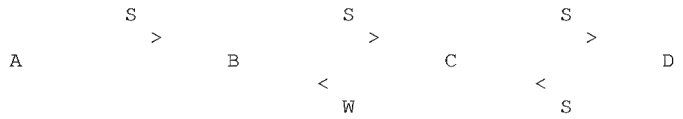
If a persistent reference is no longer needed, one should remove it from the value from which it is created. It is crucial to have unnecessary persistent references removed for efficiency, robustness and effective garbage collection.

ODStorageUnitRef is not recycled. Therefore, all ODStorageUnitRefs from a value are guaranteed to be unique.

The referring power of ODStorageUnitRef are preserved across sessions. One can write out a ODStorageUnitRef in the value, close and reopen the container and use the ODStorageUnitRef in that value to find the referred storage unit.

Strong and Weak ODStorageUnitRef

There are two kinds of ODStorageUnitRef: strong and weak. Their difference is important when a clone operation is performed (using ODDraft::BeginClone and ODDraft::EndClone). In a clone operation, all the storage units referenced by a strong ODStorageUnitRef are copied. Consider the following example,




```
S = strong persistent reference to
W = weak persistent reference to
```

The object A has a strong persistent reference to B.
The object B has a strong persistent reference to C.
The object C has a strong persistent reference to D.
The object C has a weak persistent reference to B.
The object D has a strong persistent reference to C.

If we clone A, all 4 storage units are copied.
If we clone B, only B, C and D are copied.

If we clone C, only C and D are copied.
If we clone D, only C and D are copied.

Sample Codes

Creating an auxiliary storage unit for a part:

```
void MyPart::InitPart(Environment* ev, ODStorageUnit* storageUnit)
{
  . . .

  // Create the Aux storage unit
  ODStorageUnit* auxStorageUnit =
                storageUnit->GetDraft(ev)->CreateStorageUnit(ev);

  // Add a property and value to store the persistent reference
  storageUnit->AddProperty(ev, kODPropContents);
  storageUnit->AddValue(ev, kODStrongStorageUnitRef);

  // Note that storageUnit is focused to the newly created value already.
  // Create a persistent reference to the Aux storage unit in the
  // context of the create value.
  ODStorageUnitRef ref;
  storageUnit->GetStrongStorageUnitRef(ev, auxStorageUnit, ref);

  // Note that the focus is unchanged and it is still pointing to the
  // created value.
  // Write out the persistent reference to the value.
  ODByteArray ba;
  ba._length = sizeof(ODStorageUnitRef);
  ba._maximum = sizeof(ODStorageUnitRef);
  ba._buffer = &ref;
  storageUnit->SetValue(ev, &ba);
   . . .
}
```

Let's say that the part is instantiated later in a different session, the part can retrieve its auxiliary storage unit as follows:

```
void MyPart::InitPartFromStorage(Environment* ev, ODStorageUnit* storageUnit)
{
  . . .

  // Focus the storage unit to the right value
  storageUnit->Focus(ev,
                     kODPropContents,
                     kODPosUndefined,
                     kODStrongStorageUnitRef,
                     0,
                     kODPosUndefined);

  // Retrieve the persistent reference
  ODByteArray ba;
  storageUnit->GetValue(ev, sizeof(ODStorageUnitRef), &ba);

  // Use the retrieved persistent reference to get the ID for the storage unit
  ODStorageUnitID ID =
    storageUnit->GetIDFromStorageUnitRef(ev,
                                         *((ODStorageUnitRef*) ba._buffer));

  // Dispose of the buffer
  ODDisposePtr(ba._buffer);

  // Use the ID to get the Aux storage unit from the draft.
  ODStorageUnit* auxStorageUnit =
                storageUnit->GetDraft(ev)->GetStorageUnit(ev, id);

  . . .
}
```

Storage Unit Manipulation

This section describes the basic usage of ODStorageUnit with actual examples and recipes. It is intended to supplement the class documentation. For detailed description of the ODStorageUnit methods, refer to the *OpenDoc Programming Reference* .

Recipes/Examples

Recipe 1:
Given a ODStorageUnit, create multiple properties and values.

```
su->AddProperty(ev, kODPropContents);
su->AddValue(ev, kMyKind);
su->AddValue(ev, kKindTEXT);
su->AddValue(ev, kKindPICT);
su->AddProperty(ev, kPropAnnotations);
su->AddValue(ev, kMyAnnotationsType);
```

Recipe 2:
Given the ODStorageUnit in Recipe 1, set the focus to a certain value.

There is more than one way to get to the value:

1. Focus the ODStorageUnit to the value using the known property and type.

```
su->Focus(ev,
          kODPropContents,    // Property name
          kODPosUndefined,
          kMyKind,            // Value type
          0,
          kODPosUndefined);
```

2. Focus the ODStorageUnit to the value using the known property and value index.

```
su->Focus(ev,
          kODPropContents,    // Property name
          kODPosUndefined,
          kODNULL,
          1,                  // Value index
          kODPosUndefined);
```

3. Focus the ODStorageUnit to the value using the known property and relative position.

```
su->Focus(ev,
          kODPropContents,    // Property name
          kODPosUndefined,
          kODNULL,
          0,
          kODPosFirstSib);    // Position code
```

4. Focus the ODStorageUnit to the value using a ODStorageUnitCursor. This method is especially useful when one needs to focus to a property or a value frequently.

There are two ways to create a cursor:

By specifying the focus explicitly:

```
ODStorageUnitCursor* cursor = su->CreateCursor(ev,
                                               kODPropContents,
                                               kMyKind,
```

```
0);
```

By using the focus of a storage unit:

```
ODStorageUnitCursor* cursor = su->CreateCursorWithFocus(ev);
```

One can then focus a storage unit using the created cursor:

```
su->FocusWithCursor(ev, cursor);
```

Recipe 3:
Given the ODStorageUnit from Recipe 1, iterate through all the properties.

```
// Unfocus storage unit
su->Focus(ev,
          kODNULL,
          kODPosAll,          // Position code
          kODNULL,
          0,
          kODPosAll);         // Position code

// Get the number of properties
ODULong numProperties = su->CountProperties(ev);

// Iterate through the properties
for (i = 1; i <= numProperties; i++)
{
  su->Focus(ev,
            kODNULL,
            kODPosNextSib,       // Position code
            kODNULL,
            0,
            kODPosUndefined);
}
```

Recipe 4:
Given the ODStorageUnit from Recipe 1, iterate through all the values in kODPropContents property.

```
// Focus to the desired property
su->Focus(ev,
          kODPropContents,    // Property name
          kODPosUndefined,
          kODNULL,
          0,
          kODPosAll);         // Position code

// Get the number of values
ODULong numValues = su->CountValues(ev);

// Iterate through the values
for (i = 1; i <= numValues; i++)
{
su->Focus(ev,
          kODNULL,
          kODPosSame,         // Position code
          kODNULL,
          0,
          kODPosNextSib);     // Position code
}
```

Recipe 5:
Given the ODStorageUnit from Recipe 1, remove the kPropAnnotationsProperty.

```
// Focus to the desired property
su->Focus(ev,
          kPropAnnotations,
          kODPosUndefined,
```

```
          kODNULL,
          0,
          kODPosAll);

// Remove the property
su->Remove(ev);
```

Recipe 6:
Given the ODStorageUnit from Recipe 1, remove the kKindPICT value in kODPropContents Property.

```
// Focus to the desired value
su->Focus(ev,
          kODPropContents,
          kODPosUndefined,
          kODNULL,
          kKindPICT,
          kODPosUndefined);

// Remove the value
su->Remove(ev);
```

Recipe 7:
Given the focused ODStorageUnit from Recipe 2, manipulate the data in the value. One can think of a value as a stream. In order for storage units to work in a distributed environment, all the data passed in or out through the storage unit function are in ODByteArray. The following is the structure of an ODByteArray (from the IDL emitter):

```
typedef struct
{
  unsigned long _maximum;
  unsigned long _length;
  octet *_buffer;
} _IDL_SEQUENCE_octet;

typedef _IDL_SEQUENCE_octet ODByteArray;
```

where:

_maximum contains the size of the memory block pointed to by _buffer.
_length contains the number of bytes (that is, octets) of relevant data in the memory block pointed to by _buffer. This must be smaller than or equal to _maximum.
_buffer contains a pointer to a memory block whose size is reflected in _maximum.

To add data at a particular position:

```
su->SetOffset(ev, desiredPosition);

// Set up byte array
ODByteArray ba;
ba._length = length;          // Length of data to be written
ba._maximum = maximum;        // Maximum size of buffer
ba._buffer = buffer;          // Pointer to the data
su->SetValue(ev, &ba);
```

To add data at the current offset:

```
// No SetOffset is needed.
// Just call SetValue.
su->SetValue(ev, &ba);
```

To insert data at a particular position:

```
su->SetOffset(ev, desiredPosition);
su->InsertValue(ev, &ba);
```

To append data at the end of a value:

```
// Get the length of the value
ODULong offset = su->GetSize(ev);

// Set the offset to the end of the value
su->SetOffset(ev, offset);
su->SetValue(ev, &ba);
```

To remove data from a particular position:

```
su->SetOffset(ev, offset);
su->DeleteValue(ev, length);
```

To get data from a particular position:

```
// Get to the right position
su->SetOffset(ev, offset);

// Set up byte array.
// There is really nothing to do here because the fields of
// ODByteArray are going to be filled out by GetValue.
ODByteArray ba;

// Get the data
su->GetValue(ev, desiredLength, &ba);

// Dispose the buffer created by GetValue when done
ODDisposePtr(ba._buffer);
```

Persistent Objects Removal

A persistent object has two states: a persistent state and a run-time state. The persistent state is stored in the associated storage unit while the run-time state is stored in the corresponding object instance.

Release versus Removal
When an object is released, only the run-time state is affected. For example, when the reference count of a persistent object goes down to 0 and is deleted, only the run-time state is destroyed while the persistent state remains unchanged.

However, when an object is removed, both the run-time and the persistent state are destroyed, which means the object instance and the storage unit cease to exist.

Removal without using draft function
Even though OpenDoc provides function to remove persistent objects created by the draft, a part editor does not need to remove any persistent objects using this function. This is because the underlying container suite is responsible for garbage collecting storage units that are no longer in use. A storage unit is not in use when it is no longer transitively strongly referenced by the draft properties storage unit.

Therefore, in order to remove a persistent object, all the part editor needs to do is to break or remove its reference to the storage unit of the persistent object.

```
// Focus to the value where the reference is to be removed.
partSU->Focus(...);

// Remove the storage unit ref
partSU->RemoveStorageUnitRef(ev, ref);
```

The garbage collection scheme is completely dependent on the container suite. Therefore, make sure that you write out references to the storage units of all of the desired persistent objects during externalize.

Removal using OpenDoc function
There are times when a part wants to remove a persistent object explicitly without relying on garbage collection. Here are a couple of examples:

- Two parts are tightly coupled. A part which is the creator and the sole client of a persistent object may choose to remove it at a particular time.
- A part may decide to remove a persistent object explicitly due to security reason. (However, one should note that security level provided depends on the container suite implementation).

Using the ODDraft function does not guarantee a successful removal. If the persistent object has a reference count of more than 1 when Remove is called, an exception will be thrown. It is the client's responsibility to anticipate and respond to the exception.

The following is a simple code fragment on the function usage:

```
SOM_TRY
 draft->RemovePart(ev, part);
SOM_CATCH_ALL
 // Some error handling
SOM_ENDTRY
```

Alternate Container Suite Use

The OpenDoc storage system is designed to work with multiple container suites. In addition to the bento container suite, other vendors might want to provide alternate container suites.

Creating a Container

In order to create a container, the client has to call the following ODStorageSystem method:

```
ODContainer CreateContainer(in ODContainerType containerType,
                            in ODContainerID id);
```

The first parameter of the method identifies the type of the container. In order to create a bento file container, the in parameter needs to be kODBentoFileContainer and the second parameter the FSSpec of the file where the container is to be created.

This binding is done at run-time. OpenDoc binding scans the editors folder(s) for container suite libraries. Each container suite library contains a nmap resource which lists the type(s) of containers that the container suite can create. OpenDoc binding then caches this information and uses it to determine which container suite library to use when ODStorageSystem::CreateContainer is called.

The following is the nmap resource of the bento container suite as defined in a private resource file:

```
#define kNMAPid1 128
#define kODBentoFileContainer  "OSA:Bento:ContainerType:File"
#define kODBentoMemoryContainer  "OSA:Bento:ContainerType:Memory"
#define kODFileContainerID    "ODFileContainer"
#define kODMemoryContainerID   "ODMemContainer"

// kODNameMappings, kODContainerSuite and
// kODIsAnISOString are defined by OpenDoc.
// They can be found in StdDef.r.
resource kODNameMappings (kNMAPid1)
{
  kODContainerSuite,
  {
    /* Array types: 2 elements */
    /* [1] */
    kODBentoFileContainer,
    kODIsAnISOString
    {
      kODFileContainerID
```

```
    },
    /* [2] */
    kODBentoMemoryContainer,
    kODIsAnISOString
    {
      kODMemoryContainerID
    }
  }
};
```

As it is outlined in the resource, the bento container suite can create two kinds of containers: file and memory.

Other container suite providers need to include this nmap resource into their container suite libraries. Otherwise, their container suite is never bound and created.

Acquiring a Container

The process of acquiring a container is almost the same as creating a container. The following ODStorageSystem method is used to acquire a previously created container:

```
ODContainer AcquireContainer(in ODContainerType containerType,
                             in ODContainerID ID);
```

The same process takes place to bind the container suite library for the container type supplied.

Default Container Type

The default container types are interpreted as the bento container types. Therefore, supplying kODDefaultFileContainer to AcquireContainer or CreateContainer results in a bento file container being instantiated. Similarly, supplying kODDefaultMemoryContainer to the same methods results in a bento memory container being instantiated.

Bento Container Suite

A container suite is an implementation of several storage classes (ODContainer, ODDocument, ODDraft and ODStorageUnit) which provides the main functionality of of the OpenDoc storage system. The functionality includes the maintanence of documents, drafts, persistent objects and storage units.

The bento container suite (BCS) is a container suite based on bento (a.k.a. container manager) technology.

The bento container suite currently supports two kinds of containers: file containers and in-memory containers. File containers are persistent across sessions while in-memory containers are ephemeral.

How to Create a BCS File Container?

In OpenDoc, every container is created through the ODStorageSystem.

```
ODByteArray ba;

// Set the length to the size of the file spec which
// contains meaningful data
```

```
ba._length = sizeof(short) + sizeof(long) + fsSpec->name[0] + 1;
ba._maximum = sizeof(FSSpec);
// fsSpec is the FSSpec* of the file to be opened
ba._buffer = fsSpec;

ODContainer* newContainer =
             session->GetStorageSystem(ev)->
             CreateContainer(ev,kODDefaultFileContainer,&ba);
```

How to Create a BCS In-Memory Container?

In OpenDoc, every container is created through the ODStorageSystem.

```
ODByteArray ba;
ba._length = sizeof(ODHandle);
ba._maximum = sizeof(ODHandle);

// Handle is the Handle where the bento container is going to live
ba._buffer = &handle;

ODContainer* newContainer = session->GetStorageSystem(ev)->
   CreateContainer(ev, kODDefaultMemoryContainer, ba);
```

How to Get the FSSpec from a BCS File Container?

This is not recommended because there is no way to tell whether a container is a BCS file container or not. However, if you are sure that the container is a BCS file container and you really need to know the FSSpec, here is how you do it.

```
// Get the container ID
ODContainerID containerID = container->GetID(ev);

// The returned container ID contains the FSSpec in _buffer.
// Note that the returned ID may have _length
// smaller than sizeof(FSSpec).
FSSpec* fsSpec = (FSSpec*) containerID->_buffer;

// Use it
 . . .

// Dispose the _buffer in the byte array (that is, the container ID)
ODDisposePtr(containerID->_buffer);
```

Note: This is not recommended!

Garbage Collection

The bento container suite supports garbage collection. It is triggered when a draft is saved. In order to not to lose your persistent objects, you should create references to all the persistent objects you are using during externalize.

Data Interchange

This document provides a roadmap for those who want to learn how to implement data interchange (clipboard, drag and drop and linking) in OpenDoc using the recipes.

The reader is assumed to have a basic knowledge of OpenDoc. The reader may also want to familiarize himself with the OpenDoc storage system and its recipes before proceeding.

The data interchange recipes contain the following documents:

General data interchange recipes:

- Data Interchange Basics

Drag and drop specific recipes:

- Drag and Drop

Linking specific recipes:

- Linking

Data interchange recipes which can be used for both drag and drop and clipboard:

- Clipboard Recipes
- Promises
- Drag and Drop - Promising Non-OpenDoc File

Clipboard Recipes

This document describes the basic techniques for reading and writing the Clipboard. It assumes familiarity with the document "Basic Data Interchange", and references coding examples there. For further details on the Clipboard operations involving embedded frames, see the Imaging and Layout recipes. For more information on Clipboard operations involving links, including the posting of link specifications and incorporation of linked content, see the Linking recipe.

For a description of individual methods used in the recipes, refer to the Class Documentation for the appropriate class.

About Coding Examples

This document contains a number of coding examples in addition to descriptive text. The coding examples cover:

MyWriteToClipboard	Writing intrinsic content to the clipboard.
MyCloneEmbeddedFrameToClipboard	Writing a single embedded frame to the clipboard.
MyIncorporateFromClipboard	Incorporating content from the clipboard.
MyEmbedFromClipboard	Embedding content from the clipboard.

For simplicity, the coding examples use ODByteArrays to represent the content to be read or written. If your part does not maintain its content as an ODByteArray, you can either copy it into an ODByteArray, or, more likely, write data out or read data in chunks using the StorageUnitSetValue or StorageUnitGetValue utility functions.

The examples use exception handling in the form of TRY...CATCH_ALL...ENDTRY blocks to catch exceptions, and assumes that errors returned by SOM methods are automatically thrown upon return.

Note: Code appearing in these recipes has not been tested, and may contain errors.

Header File

Parts that access the clipboard need to include the clipboard header file:

```
#ifndef SOM_ODClipboard_xh
#include <Clipbd.xh>
#endif
```

When Parts May Access the Clipboard

Parts should access the clipboard only when two conditions are true. First, the part must be running in the front-most process. The Clipboard is only defined for the front-most process; any attempt to access it in a background process is meaningless (the clipboard method Clear now returns error kODErrBackgroundClipboardClear if called when the process is in the background). Second, the part must own the clipboard focus.

In addition, note that if a part's draft is read only, it should disable Clipboard operations that would change the draft, such as Cut, Paste, and Paste As.

Except for the special case of AdjustMenus, described below, parts should complete their access to the clipboard before returning from the method that initiated the access. For example, parts should modify the clipboard within their HandleEvent method, and relinquish the clipboard focus before returning from HandleEvent. Parts should not spawn a thread that copies data to the clipboard, all the while holding the clipboard focus. Parts can minimize the amount of data transferred on a Cut or Copy by writing promises to the clipboard.

Determining that a Draft Can Be Modified

If a part calls an OpenDoc method that modifies a draft, and the draft's permissions do not allow modifications, the OpenDoc method will return an error. Parts can determine if a draft can be modified by testing the draft permissions:

```
ODDraftPermissions permissions = storageUnit->GetDraft(ev)->GetPermissions(ev);

if (permissions >= kDPSharedWrite)
{
 ... the draft may be modified
}
```

Parts should check draft permissions before attempting an operation that modifies a draft. For example, a part should not enable the Cut, Paste, or Paste As menu items if the draft cannot be modified.

Clipboard Focus

To be thread safe, parts should acquire the clipboard focus prior to accessing the clipboard. Since parts usually need to inspect the clipboard to enable items on the Edit menu, they should call Arbitrator::RequestFocus from AdjustMenus to request the clipboard focus. If granted, a part can enable the Cut, Copy, Paste, and Paste As items as appropriate. A part can assume that a call to AdjustMenus will be followed by a call to HandleEvent; most parts should request the clipboard focus in AdjustMenus and relinquish the focus in HandleEvent.

Acquiring the Clipboard Focus

```
ODSession* session = somSelf->GetStorageUnit(ev)->GetSession(ev);
ODArbitrator* arbitrator = session ->GetArbitrator(ev);

ODTypeToken clipboardFocus = session->Tokenize(ev, kODClipboardFocus);
ODFrame* clipboardOwner = arbitrator->AcquireFocusOwner(ev, clipboardFocus);
if ((frame == clipboardOwner) ||
    arbitrator->RequestFocus(ev, clipboardFocus, frame))
{
  ODClipboard* clipboard = session->GetClipboard(ev);
  // Access the clipboard here.
  ...
}
ODReleaseObject(ev, clipboardOwner);
```

Parts may retain the clipboard focus while active, but must be prepared to relinquish the clipboard focus to allow other parts to inspect the clipboard. Parts must also relinquish the clipboard focus in their DeactivateFrame method if the deactivated frame owns the focus.

Relinquishing the Clipboard Focus

```
// Insert in your part's DeactivateFrame method (only necessary if
// the part holds the Clipboard focus)

ODSession* session = somSelf->GetStorageUnit(ev)->GetSession(ev);
ODArbitrator* arbitrator = session ->GetArbitrator(ev);

arbitrator->RelinquishFocus(ev, clipboardFocus, frame);

// Insert into your part's RelinquishFocus method

ODSession* session = somSelf->GetStorageUnit(ev)->GetSession(ev);
ODArbitrator* arbitrator = session ->GetArbitrator(ev);

if (focus == clipboardFocus)
  arbitrator->RelinquishFocus(ev, clipboardFocus, frame);
```

Cloning to the Clipboard

Whenever a part puts data on the clipboard, even if object cloning is not performed, it should do so within a BeginClone-EndClone transaction (using the clone kind kODCloneCut or kODCloneCopy, as described in the Data Interchange Basics document). Using a clone transaction helps OpenDoc ensure that unfulfilled promises are resolved when a draft is closed.

Note that the first paste following a cut is always a move in OpenDoc. There is no way for the part performing a paste to force pasting a copy of the data. This should not matter to your part; the paste recipes work whether the content is being moved or copied into your part. However, the behavior of links and embedded content will be different when moved versus copied.

Clipboard Update IDs

The update ID returned by the ActionDone and GetUpdateID methods of the clipboard object identifies a particular clipboard generation, not a particular change. When your part pastes data from the clipboard, your part must remember the clipboard update ID in case the operation is later undone, but your part should generate a new update ID (using the UniqueUpdateID method of the session object) to be associated with the pasted content. If the user pastes the contents of the clipboard twice, the clipboard's update ID will be the same both times. Each paste is a separate change, however, with which your part must associate different update IDs. See the Linking recipe for more information on update IDs.

Undoing Clipboard Operations

Parts that support Cut, Paste, Drag or Drop must support undoing and redoing those operations. If the user moves frames, links, or other objects from one part to another, via Cut and Paste or Drag and Drop, objects may be reused at the destination. If the clipboard or Drag and Drop operation is undone, the objects must be reinstated at the source. This can only happen if the part initiating the cut or drag and the part performing the paste or drop both support undo.

Your part is responsible for notifying the clipboard whenever a Cut, Copy, or Paste operation is done, undone, or redone. When your part performs a Cut, Copy, or Paste, call the ActionDone method of the clipboard. This method takes as its argument the cloneKind your part used to access the clipboard (either kODCloneCut, kODCloneCopy, or kODClonePaste). When your part undoes a Cut, Copy, or Paste, call the ActionUndone method of the clipboard object, specifying the update ID returned by the clipboard's ActionDone method at the time of the original action, and the clone kind used at the original access. When your part redoes a Cut, Copy, or Paste, call the ActionRedone method of the clipboard object instead.

When a cut or copy operation is undone, it is not necessary to restore the clipboard to its previous contents. Because of this, code implementing the redo of a paste operation cannot assume the clipboard content is the same as it was when the original paste was performed. Redo of a paste operation must be implemented by restoring content from a private cache.

When a part cuts an object to the clipboard, a reference to the object should be saved in an undo action. If the part's UndoAction method is called, it must (1) reinstate the object into its content model from its undo information, and (2) notify the clipboard that the cut was undone (see below). Similarly, if the part's RedoAction method is called, it should (1) remove the object from its content model, and (2) notify the clipboard that the cut was redone. When the part's DisposeActionState method is called, the object reference in the action data should be released. See the section below for special instructions on handling embedded frames that have been cut or pasted.

Undoing the Cut or Paste of Embedded Frames

The part performing a cut or paste of one or more embedded frames is responsible for handling the in-limbo status of the frame correctly. If the user undoes the cut of an embedded frame, your part must set the in-limbo flag to kODFalse in its UndoAction method. If the user redoes the cut of an embedded frame, your part must set the in-limbo flag to kODTrue. When your DisposeActionState method is called to commit a cut, your part must examine the state of the in-limbo flag. If the IsInLimbo returns kODFalse, your part must release the frame reference it holds; if IsInLimbo returns kODTrue, your part must remove the frame by calling the Remove method of the frame.

Similarly, if the user undoes the paste of an embedded frame, your part's UndoAction method must restore the in-limbo flag to the value it had before the paste was performed. If the user redoes the paste of an embedded frame, your part must set the in-limbo flag to kODFalse. When your DisposeActionState method is called to commit a paste, your part must examine the state of the in-limbo flag. If the IsInLimbo returns kODFalse, your part must release the frame reference it holds. If IsInLimbo returns kODTrue, your part's action depends on the in-limbo status of the frame at the time the paste was performed. If the frame was in-limbo, your part must release the frame reference it holds (some other part will remove it.) If the frame was not in-limbo, your part must remove the frame by calling the Remove method of the frame.

Cleanup in ReleaseAll

When a part's ReleaseAll method is called, the part should check to see if it has a promise or a link specification on the clipboard. If so, it needs to fulfill the promise or remove the link specification.

Writing Intrinsic Content to the Clipboard

MyWriteToClipboard demonstrates how a part might implement a routine to copy intrinsic content to the clipboard. It uses the MyWriteToContentSU method, described in the Data Interchange Basics recipe, to write promises for the data. MyWriteToClipboard should only be called when this part holds the clipboard focus and is in the foreground, as described earlier.

The promiseData parameter is whatever information the part needs to identify the promised content when its FulfillPromise method is called. The contentShape parameter will be written as the suggested frame shape annotation. This property will be used as the frame shape should content be embedded from the clipboard.

The partName parameter will be written as the part name annotation. If your part associates a name with the content written, you may want

the part created should the content be embedded at the destination to bear this name. If not, the partName may be null, and no part name annotation will be created.

The optional linkSpecData parameter will be written into a link specification if the operation is a copy.

The cloneKind parameter is either kODCloneCut or kODCloneCopy.

This method returns the update ID of this clipboard operation. This ID can be saved and later compared to the current clipboard update ID to determine if the contents of the clipboard have been replaced.

```
ODUpdateID MyWriteToClipboard (Environment *ev,
  ODByteArray* promiseData,
  ODShape* contentShape,
  ODIText* partName,
  ODByteArray* linkSpecData,
  ODCloneKind cloneKind)
{
  ODClipboard* clipboard = fSOMSelf->
                           GetStorageUnit(ev)->
                           GetSession(ev)->
                           GetClipboard(ev);
  ODVolatile(clipboard);

  // Remove any existing data on the clipboard
  clipboard->Clear(ev);

  // Get the content storage unit for the clipboard
  ODStorageUnit* clipContentSU = clipboard->GetContentStorageUnit(ev);

  TRY

    self->MyWriteToContentSU(ev, clipContentSU, cloneKind, promiseData,
                             contentShape, partName);

    // If a copy operation is being performed,
    // write a link spec to the clipboard.
    if ((cloneKind == kODCloneCopy) && linkSpecData)
    {
      // See Linking.
      ...
    }

  CATCH_ALL

    // If an exception was raised, clear the clipboard.
    //   The previous contents of the clipboard are lost.
    clipboard->Clear(ev);
    RERAISE;

  ENDTRY

  // Return the update ID associated with this clipboard operation.
  return clipboard->ActionDone(ev, cloneKind);
}
```

Copying Intrinsic Content to the Clipboard

Copying intrinsic content to the clipboard is as simple as calling MyWriteToClipboard with the suitable clone kind.

```
ODUpdateID clipboardUpdateID = self->MyWriteToClipboard(ev,
    promiseData,
    contentShape,
    partName,
    linkSpecData,
    kODCloneCopy);
```

Cutting Intrinsic Content to the Clipboard

If the operation is a cut, after putting content on the clipboard, the intrinsic content should be removed. When an embedded frame is cut, the part must delete any facets displaying that frame. Call the containing facet's RemoveFacet method to remove a facet, then delete the facet object. The cut frame's containing frame should be set to kODNULL to remove it from the layout hierarchy, but not removed from the part. A reference to the cut frame should be placed in undo action data. Other frames displaying the part are not affected.

When content is cut, any objects cloned to the clipboard may be reused if pasted back into the same draft. This includes embedded frames, link and link source objects that a part typically references directly. If the cut content references any other persistent objects, those references should retained in undo data so the operation can be undone. When the references are no longer needed to support undo, the objects should be released. See Undoing Clipboard Operations for more information.

Each embedded frame cut along with intrinsic content must have its in-limbo flag set to kODTrue.

```
cutEmbeddedFrame->SetInLimbo(ev, kODTrue);
```

Copying a Single Embedded Frame to the Clipboard

Your part should follow this recipe to put a single embedded frame on the clipboard. The promiseProxyData parameter. Note the similarity to MyWriteToClipboard; only the line calling MyCloneEmbeddedFrameToContentSU is different!

The embeddedFrame parameter is the frame being cut or copied. The optional promiseProxyData is proxy content that this part associates with the embedded frame and will be written as proxy content. The optional linkSpecData parameter will be written into a link specification if the operation is a copy. The cloneKind parameter is either kODCloneCut or kODCloneCopy.

This method returns the update ID of this clipboard operation.

```
ODUpdateID MyCloneEmbeddedFrameToClipboard (Environment *ev,
  ODFrame* embeddedFrame,
  ODByteArray* promiseProxyData,
  ODByteArray* linkSpecData,
  ODCloneKind cloneKind)
{
  ODClipboard* clipboard = somSelf->
                           GetStorageUnit(ev)->
                           GetSession(ev)->
                           GetClipboard(ev);
  ODVolatile(clipboard);

  // Remove any existing data on the clipboard
  clipboard->Clear(ev);

  // Get the content storage unit for the clipboard
  ODStorageUnit* clipContentSU = clipboard->GetContentStorageUnit(ev);

  TRY

    self->MyCloneEmbeddedFrameToContentSU(ev, clipContentSU, cloneKind,
                                          embeddedFrame, promiseProxyData);

    // If a copy operation is being performed, write a link spec to the
    //   clipboard if this part supports linking.
    if ((cloneKind == kODCloneCopy) && linkSpecData)
    {
      // See Linking; note conditions when
      // a link specification should not be written.
      ....
    }
    clipboard->ExportClipboard(ev);

  CATCH_ALL

    // If an exception was raised, clear the clipboard.
    // he previous contents of the clipboard are lost.
    clipboard->Clear(ev);
    RERAISE;

  ENDTRY

  // Return the update ID associated with this clipboard operation.
```

```
  return clipboard->ActionDone(ev, cloneKind);
}
```

Cutting a Single Embedded Frame to the Clipboard

Unlike Copy, Cut is an undoable operation. Your implementation of Cut can be based on Copy, with a little work before and after the copy.

```
// Save the current selection in an undo action.  MyMakeUndoAction is not
//  described here.
ODActionData actionState = somSelf->MyMakeUndoAction(ev);

ODUpdateID clipboardUpdateID =
    self->MyCloneEmbeddedFrameToClipboard(embeddedFrame, promiseData,
                                          kODCloneCut);

// Set the in-limbo flag of the cut frame to kODTrue.
embeddedFrame->SetInLimbo(ev, kODTrue);

// Add an undo action for this operation and delete the selection.
ODUndo* undo = somSelf->GetStorageUnit(ev)->GetSession(ev)->GetUndo(ev);
undo->AddActionToHistory(ev, somThis->fPartWrapper, actionState,
                         kODSingleAction, ...);

// MyDeleteSelection is a part-specific method not described here.
somSelf->MyDeleteSelection(ev);

// Associate an update ID with this cut, and notify containing parts of this change
ODUpdateID cutUpdateID = fSOMSelf->
                         GetStorageUnit(ev)->
                         GetSession(ev)->
                         UniqueUpdateID(ev);
myFrame->ContentUpdated(ev, cutUpdateID);

// Make sure that this change is saved when the document is closed.
somSelf->GetStorageUnit(ev)->GetDraft(ev)->SetChangedFromPrev(ev);
```

Incorporating Content from the Clipboard

Incorporating from the clipboard is demonstrated by this code that gets the clipboard content storage unit and calls the MyReadFromContentSU routine described in the Data Interchange Basics document. In this simple example, the incorporated content is inserted where there are no source links maintained by this part. Because Paste must be an undoable operation, action data is added to the undo history if incorporation is successful.

The targetFrame argument identifies the frame being incorporated into and is passed to MyReadFromContentSU.

This method returns the update ID of this clipboard operation. (This is not the same as the update ID associated with the content change to this part!)

```
ODUpdateID MyIncorporateFromClipboard (Environment *ev, ODFrame* targetFrame)
{
  ODClipboard* clipboard = fSOMSelf->
                           GetStorageUnit(ev)->
                           GetSession(ev)->
                           GetClipboard(ev);
  ODUpdateID clipboardUpdateID = kODUnknownUpdate;

  // Get the content storage unit of the clipboard
  ODStorageUnit* clipContentSU = clipboard->GetContentStorageUnit(ev);

  ODVolatile(clipboard);

  TRY
    fSOMSelf->MyReadFromContentSU(ev,
```

```
                                clipContentSU,
                                targetFrame,
                                kODClonePaste,
                                kODNotInLink);

    clipboardUpdateID = clipboard->ActionDone(ev, kODClonePaste);

    // If the incorporation was successful, add an undo action.
    // The undo data should include clipboardUpdateID and
    // the in-limbo status of any frames that were embedded.
    ODActionData actionState = ... // part specific
    ODUndo* undo = somSelf->GetStorageUnit(ev)->GetSession(ev)->GetUndo(ev);
    undo->AddActionToHistory(ev,
                             somThis->fPartWrapper,
                             actionState,
                             kODSingleAction,
                             ...);

    // Associate an update ID with this paste,
    // and notify containing parts of this change
    ODUpdateID pasteUpdateID =
                fSOMSelf->GetStorageUnit(ev)->GetSession(ev)->UniqueUpdateID(ev);
    targetFrame->ContentUpdated(ev, pasteUpdateID);

    // Make sure that this change is saved when the document is closed
    fSOMSelf->GetStorageUnit(ev)->GetDraft(ev)->SetChangedFromPrev(ev);

  CATCH_ALL

  ENDTRY

  // Return the update ID associated with this clipboard operation.
  return clipboardUpdateID;
}
```

Embedding a Part from the Clipboard

Embedding from the clipboard is demonstrated using the routine MyEmbedContentSU, which performs the actual work and is described in the Data Interchange Basics document.

The targetFrame argument identifies the frame being incorporated into and is passed to MyEmbedContentSU.

This method returns the update ID of this clipboard operation.

```
ODUpdateID MyEmbedFromClipboard (Environment *ev, ODFrame* targetFrame)
{
  ODClipboard* clipboard = fSOMSelf->
                           GetStorageUnit(ev)->
                           GetSession(ev)->
                           GetClipboard(ev);

  // Get the content storage unit for the clipboard
  ODStorageUnit* clipContentSU = clipboard->GetContentStorageUnit(ev);

  ODVolatile(clipboard);

  TRY

    fSOMSelf->MyEmbedContentSU(ev,
                               clipContentSU,
                               targetFrame,
                               kODClonePaste,
                               kODNotInLink);

    // Associate an update ID with this paste,
    // and notify containing parts of this change
    ODUpdateID pasteUpdateID = fSOMSelf->
                               GetStorageUnit(ev)->
                               GetSession(ev)->
                               UniqueUpdateID(ev);
    targetFrame->ContentUpdated(ev, pasteUpdateID);

    // Make sure that this change is saved when the document is closed
```

```
    fSOMSelf->GetStorageUnit(ev)->GetDraft(ev)->SetChangedFromPrev(ev);

  CATCH_ALL

  ENDTRY

  // Return the update ID associated with this clipboard operation.
  return clipboard->ActionDone(ev, kODClonePaste);
}
```

Container Application Requirements

Container applications need to call clipboard object methods to ensure synchronization with the host platform clipboard mechanism, and to inform the clipboard of draft closings so the correct move or copy semantics can be enforced. Parts do not call these methods.

The ExportClipboard method must be called by the Container Application whenever the current content of the OpenDoc clipboard must be transferred to the platform clipboard. This includes when the application is suspended or quit. On the OS/2 platform, this is currently done by parts whenever they change the content of the clipboard object.

The DraftSaved method must be called to notify the clipboard object that a document draft has been saved.

The DraftClosing method must be called to notify the clipboard object that a document draft is closing. This call must be made before the draft object is closed.

Data Interchange Basics

This document contains recipes for transferring data between OpenDoc parts and data interchange objects. These basic recipes are referenced by the specific recipes for each data interchange object: Clipboard recipes, Drag and Drop recipes, and Linking recipes. The recipes here assume familiarity with the OpenDoc Storage recipes. Other OpenDoc recipes that may be of interest to developers implementing data interchange include the Info Recipes. For descriptions of individual methods used in the recipes, refer to the Class Documentation.

Note: Code appearing in these recipes has not been tested, and may contain errors. They are provided to help you get started with data interchange, but need to be customized for your particular content model.

About Coding Examples

This document contains a number of coding examples in addition to descriptive text. The coding examples cover:

MyCloneStrongReference MyCloneWeakReference	How to clone objects referenced by your part.
MyWriteToContentSU	Writing intrinsic content to a data interchange object.
MyCloneEmbeddedFrameToContentSU	Writing a single embedded frame to a data interchange object.
CloneInto	Implementing your part's CloneInto method.
MyReadFromContentSU	Incorporating content from a data interchange object.
MyEmbedContentSU	Embedding content from a data interchange object.
MyMakeEmbeddedFrame	How your part might make a new frame for embedded content.

The coding examples appearing in this document make use of utility routines in addition to public OpenDoc functions. The reader is referred to

the utilities documentation for information about specific utilities.

For simplicity, the coding examples use ODByteArrays to represent the content to be read or written. If your part does not maintain its content as an ODByteArray, you can either copy it into an ODByteArray, or, more likely, read and write data in chunks using the StorageUnitSetValue or StorageUnitGetValue utility functions.

The examples use exception handling in the form of TRY...CATCH_ALL...ENDTRY blocks to catch exceptions, and assumes that errors returned by SOM methods are automatically thrown upon return.

To ensure reference-counted objects are released when exceptions are raised, some of these examples employ temporary objects which encapsulate reference-counted objects and automatically release them when their scope is exited.

Data Interchange Mechanisms

The exchange of content within and between parts in OpenDoc involves the use of three mechanisms: the Clipboard, Drag and Drop, and Linking. While each mechanism has its own characteristics, much of the low-level details are the same. This document discusses those common details, and provides examples of how your part can be organized to leverage this commonality.

The routines shown here are the workhorses of data interchange. They do the work of moving content to and from a data interchange object. Once your part has these in place, implementing Clipboard support in you part, for example, becomes the simpler problem of gaining access at the appropriate time to the clipboard content storage unit. The same for Drag and Drop and Linking, although with Linking you need to merge the persistent Linking objects into your content model as well.

Content Storage Units

At the lowest level, data interchange involves writing content to, and reading content from, one of these data interchange objects: the Clipboard object, the Drag and Drop Object, a Link Source object, or a Link object. LinkSource and Link objects also happen to be persistent objects. These four objects have similar interfaces, but these similarities are not based on class inheritance.

Each data interchange object has a distinguished storage unit, called the content storage unit, used to exchange content. Access to this storage unit is provided by a GetContentStorageUnit method defined by all data interchange objects. The content storage unit serves as the root of the storage units that collectively define the content of the interchange object. Parts can use the same code to write to the content storage unit of any data interchange object.

It is important to distinguish between a content storage unit and a persistent object storage unit. Links and LinkSource objects, for example, are data interchange objects that also happen to be persistent objects. All persistent objects have storage units which stores their persistent state; Links and LinkSource objects have a storage unit containing their persistent state. This is NOT the content storage unit, and your part should never read from or write to this storage unit. Your part must use a content storage unit to exchange data.

Content Storage Units are Part Storage Units

Any content storage unit can be considered a part storage unit (although content storage units are never bound to a part editor). The content storage unit of the clipboard is always a part. The content storage unit of a Drag and Drop object is always a part. The content storage unit of a link is always a part. All content storage units have the only property required for a part: a kODPropContents property. (Exception: The content storage unit of a link may be missing the kODPropContents property if updating the link failed; see the Linking for more information about this situation if your part supports linking. Likewise, the content storage unit of the Clipboard may also be missing the kODPropContents property if nothing has be cut or copied to the clipboard. Before cloning a content storage unit from the Clipboard, check to ensure the kODPropContents property is present.) Parts writing to a content storage unit must ensure they follow the same rules as for any part storage unit, for example, writing all data in the kODPropContents property. This enables a part to embed a content storage unit by simply cloning it; the destination does not have to "construct" a part storage unit.

Data Interchange Properties

These properties are relevant to data interchange and may appear in a content storage unit. Except for the kODPropContents property, these properties must be read directly from a content storage unit because they are not copied when a content storage unit is cloned.

kODPropContents
: This property contains the data being transferred. It is in the same format as in a part's storage unit. We recommend that your part write a promise for data interchange, so only data used by a destination is actually transferred.

kODPropProxyContent
: This optional property may appear when a single embedded frame is written to the content storage unit. This property also contains content, but content that the part initiating the data transfer associates with the embedded frame (like a drop shadow or positional information). Like kODPropContents, this may be promised, too.

  Proxy content is NOT copied when a destination part clones a content storage unit to embed the content in its draft. Destinations must read proxy content from a content storage unit, not from the storage unit cloned from the content storage unit.

kODPropCloneKindUsed
: This property may be present in a content storage unit to indicate the clone kind to use in fulfilling a promise in the storage unit. This property is created by a part cloning a single embedded frame to a data interchange object. It may also be created by any part writing a promise to a data interchange object. This property should contain the value kODCloneCut, kODCloneCopy, or kODCloneToLink.

  The clone kind used property is NOT copied when a content storage unit is cloned.

kODPropContentFrame
: This property references a frame storage unit for the data interchange content. This property should only appear when a single embedded frame is written to the content storage unit.

  The content frame property is NOT copied when a destination part clones a content storage unit to embed the content in its draft. Destinations must read proxy content from a content storage unit, not from the storage unit cloned from the content storage unit.

kODPropSuggestedFrameShape
: This property contains a suggested frame shape for use when the content is embedded rather than incorporated at the destination. This should be written by parts initiating a data transfer that write intrinsic content. A frame shape should not be written in addition to a kODPropContentFrame property. The shape applies to all representations in the kODPropContents property. If neither the kODPropSuggestedFrameShape nor the kODPropContentFrame property exists, the destination part will have to use a default shape for the embedded frame.

  The frame shape written to the kODPropSuggestedFrameShape property does not specify the location of the shape in the original content. Positional information, in general, is relevant only when the content is pasted into a part of the same category. In this case, positional information is included as part of the content representation. If the content is embedded, its location should be determined by the destination part.

  The suggested frame shape property is NOT copied when a destination part clones a content storage unit to embed the content in its draft. Destinations must read proxy content from a content storage unit, not from the storage unit cloned from the content storage unit.

kODPropLinkSpec
: This optional property is present when a link may be created to the source of the content in the kODPropContents property. See the Linking recipe for information content linking in OpenDoc and on link specifications.

  The link specification is NOT copied when a destination part clones a content storage unit to embed the content in its draft. Destinations that wish to create a link must read the link spec from a content storage unit, not from the storage unit cloned from the content storage unit.

kODPropMouseDownOffset
: In a drag and drop operation, the part initiating the drag can add this property to the content storage unit to indicate the offset from the mouse down point to the beginning of the selection, so the part receiving the drop can align the result of the drop with the dragged image, if desired.

  The mouse down offset is NOT copied when a destination part clones a content storage unit into its draft.

All Data In the Contents Property

All data maintained by your part should be stored in values in the contents property of your storage unit. Similarly, when write data to a content storage unit, you should only be writing to the contents property. OpenDoc does not expect you to store data in other properties, and such properties are subject to removal. That said, your part is free to create as many auxiliary storage units as you like, with whatever properties and kinds you like. However, you must access these auxiliary storage units from a reference stored in one or more values in your contents property.

Moreover, each value in a contents property should be a self-sufficient representation of your data. Some values may be lower fidelity representations, such as a plain text representation of a styled text part. Values may reference common auxiliary storage units, but it should not be necessary to read multiple values to assemble a representation of your data.

You are expected to order the values in a contents property in order of decreasing fidelity. Ordering is determined when the value type is added to the property. Add or promise the highest fidelity value type first.

How Many Kinds Are Enough?

The number of content kinds your part writes to a content storage unit is entirely up to your part. In general, your part should write the kind that is the current part kind (not necessarily the highest fidelity kind your part supports), plus one or more standard interchange formats depending on your category. This is in general a good compromise between size and utility. Most destinations will be able to accept the data, and you will not be writing too many redundant copies of your data. Remember that OpenDoc includes a translation facility that destinations can use to provide your data in formats you do not write or do not support, albeit at a potential loss of fidelity.

Promise what you Have - Deliver what Is Needed

We recommend that parts promise content whenever possible when writing to a data interchange object. That way, only the kinds actually used need to be transferred. The recipes here demonstrate writing promises. This also helps clarify the recipes, since the delivery of part-specific content is pushed into your part's FulfillPromise method.

When writing a promise in your part's CloneInto method (as described later on in a recipe), a part should determine which of its embedded frames are included in the frame scope, and record those frames as part of the promise data it writes. The part's FulfillPromise method can then supply the correct content. Note that a part must fulfill a promise with the content present when the promise was written, even if the original content is changed before the promise is fulfilled.

It is up to your part to ensure you can fulfill outstanding promises it has written. For example, if your part writes promises to the clipboard, you must be able to fulfill those promises for an indefinite period of time. Note that your part is not informed when promises are removed. Your part may need to preemptively fulfill clipboard promises should it become impossible for your part to fulfill them in the future. For example, if your part fulfills promises using undo action data, be aware that your part's DisposeActionState may be called while a promise still exists on the Clipboard, so your part may need to fulfill the promise at that time.

When writing a promise, make sure your FulfillPromise method will have the following information:

- If the promise is written by your part's CloneInto method, the display frame(s) of your part as identified by the scope parameter.
- If your part initiated the data transfer, add a kODPropCloneKindUsed property to the storage unit so the correct ODCloneKind constant can be specified when fulfilling the promise.
- Some means of identifying the content that must be delivered.

Note that your part may write multiple promises for cut content (one promise for each value type). Your part's FulfillPromise method does not need to behave differently the first time it fulfills a promise for the cut content, verses fulfilling a second promise for the same content. OpenDoc will ensure that the first time cut data is pasted, the operation is a move, while each subsequent paste behaves like a copy. Your FulfillPromise can clone using the ODCloneKind value in the kODPropCloneKindUsed property each time. Any special action required by the cut must be carried out when the promise was written or in the course of handling an Undo action for the cut.

Exchanging Intrinsic Versus Embedded Content

Most data transfers involve writing or reading intrinsic content. Intrinsic content is content managed by the part doing the data transfer; for example, a Text part putting selected text on the clipboard is writing intrinsic content. If the selection includes an embedded picture as well as text, that too is considered writing intrinsic content. In both cases, the part initiating the data transfer is writing content to a content storage unit.

The one exception to transferring intrinsic content occurs when a single embedded frame is involved. In this case, the content of primary importance is that in the embedded frame, not in the containing part initiating the transfer. When a single embedded frame is selected, parts follow different recipes during data interchange.

Making the Embed Versus Incorporate Decision at a Destination

As long as it has a kODPropContents property, a content storage unit is a part, and can be either incorporated or embedded at a destination. When a part handles a Paste or Paste As command, or receives a drop, it must decide whether to embed or incorporate the data (simple parts that do not support embedding will just incorporate).

If the content storage unit contains a kODPropContentFrame property, the data was cut, copied, or dragged as a single embedded frame. It should be embedded at the destination, even if the data could be incorporated.

Otherwise, on paste or drop the part should incorporate the data if it makes sense according to its content model. The user can override the part's default behavior by using the Paste As command.

Choosing the Content Kind to Incorporate

When incorporating data during a paste or drop, a part must decide which data format to use. When incorporating content, your part is free to use the highest fidelity kind in the content storage unit that it supports. It is not necessary to incorporate the preferred kind specified by a kODPropPreferredKind property, if present.

Cloning

Persistent objects are transferred by a process called cloning. Your part usually references other persistent objects like embedded frames or link objects. If your part initiates a transfer of intrinsic content that references other objects, your part transfers these objects by cloning them. Similarly, your part's CloneInto method is called when your part is involved in a data transfer operation initiated by another part. Cloning is described in more detail in the Cloning Overview document.

Clone Kinds Used To Initiate a Clone Transaction

Cloning is performed within a transaction started by calling ODDraft::BeginClone and completed by calling ODDraft::EndClone or ODDraft::AbortClone. The BeginClone method takes as one of its parameters an ODCloneKind value. When a part initiates a cloning transaction, its important to specify the clone kind constant appropriate for the semantics of the operation. This indication is necessary in order for OpenDoc to maintain the behavior defined by the OpenDoc Human Interface Specification when their underlying content is copied or moved. The value kODCloneCopy informs OpenDoc that the clone is into an intermediate draft (like the clipboard or a drag and drop container) with copy semantics; kODCloneCut implies cut semantics. The value kODClonePaste informs OpenDoc that the clone is from an intermediate draft into a destination draft. Other ODCloneKind values are used when cloning to and from links; see the Linking for more

information.

Cloning Referenced Objects

Your part moves a referenced object to and from a data interchange object by cloning the object. You clone an object by specifying its object ID; if the object has been internalized, the object will clone itself via its CloneInto method, otherwise, the object's storage unit will perform the clone.

Your part distinguishes between strong and weak references. Strong references are those to objects that must also be cloned. Weak references are backward links that your part uses but which may be broken during data interchange. Weak references are important because they prevent irrelevant objects from being copied during a clone operation. When your part clones a strongly or weakly referenced object, its your part's responsibility to use Clone or WeakClone as appropriate.

The recipes below show how your part should implement cloning an object from a reference. Important points to remember:

- The methods below must be called during a clone transaction, identified by the draftKey parameter.
- The example cloning methods shown below take a parameter specifying the object to be cloned, but do not take a parameter specifying the storage unit to clone into. Its rare that your part needs to clone into a specific storage unit; the only typical situation is when your part explicitly clones an embedded part into the content storage unit of a data interchange object. Note that if your part clones an object twice during the same cloning transaction (or when fulfilling a promise) and kODNULLID is specified as the destination storage unit, the same destination will be automatically used the second time.
- Never assume a storage unit reference is valid, not even a strong reference. Test the reference using IsValidStorageUnitRef before calling GetIDFromStorageUnitRef to get the referenced object's ID.
- The object ID returned by these methods should not be used to internalize the persistent object until after the clone transaction completes successfully. Your part must either hold on to the object IDs, or convert them to references. If your part did not start the clone transaction, you need to test the object ID for validity before creating a reference to it. If IsValidID returns false, you need to remove the object from the data you are writing, because you will not be able to create the reference. (GetStrongStorageUnitRef and GetWeakStorageUnitRef fail if called with an invalid object ID.)

```
SOM_Scope ODID  SOMLINK MyPartMyCloneStrongReference(MyPart
                                                     *somSelf,
                                                     Environment *ev,
  ODStorageUnit* su,
  ODStorageUnitRef suRef,
  ODDraftKey draftKey)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","MyCloneStrongReference");

  ODID clonedID = kODNULLID;

  if (su->IsValidStorageUnitRef(ev, suRef))
  {
    SOM_TRY

      ODID storageUnitID = su->GetIDFromStorageUnitRef(ev, suRef);
      clonedID = su->GetDraft(ev)->Clone(ev,
                                         draftKey,
                                         storageUnitID,
                                         kODNULLID,
                                         kODNULLID);

    SOM_CATCH_ALL

    SOM_ENDTRY
  }

  return clonedID;
}

SOM_Scope ODID SOMLINK MyPartMyCloneWeakReference(MyPart
                                                  *somSelf,
                                                  Environment *ev,
  ODStorageUnit* su,
  ODStorageUnitRef suRef,
  ODDraftKey draftKey)
```

```
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","MyCloneWeakReference");

  ODID clonedID = kODNULLID;

  if (su->IsValidStorageUnitRef(ev, suRef))
  {
    SOM_TRY

      ODID storageUnitID = su->GetIDFromStorageUnitRef(ev, suRef);
      clonedID = su->GetDraft(ev)->WeakClone(ev, draftKey, storageUnitID,
                                             kODNULLID, kODNULLID);

    SOM_CATCH_ALL

    SOM_ENDTRY
  }

  return clonedID;
}
```

Specifying Platform Kinds

If your part supports a standard platform data format, such as 'TEXT' or 'PICT' data, your part can determine the ODValueType designating that kind using the GetISOTypeFromPlatformType method of the translation object. This example shows how to specify the ODValueType for plain text. Supporting a platform data type allows your application to interchange data with non-OpenDoc applications supporting that type.

```
ODTranslation* translation = mySession->GetTranslation(ev);
ODValueType isoTypeAppleScrapTEXT =
    translation->GetISOTypeFromPlatformType(ev,
                                            ".TEXT",
                                            kODPlatformDataType);
```

Promising Intrinsic Content

This example demonstrates a method that can be used to write the content storage unit of a link, the clipboard, or drag and drop object. This routine can be used only to promise intrinsic data; if a single embedded frame is selected, you should follow Writing an Embedded Frame.

When promising intrinsic content, your part typically writes two properties, kODPropContents and kODPropSuggestedFrameShape, and optionally a kODPropPartName. This is all the content storage unit needs to be either incorporated or embedded at the destination.

Your part may write a kODPropPartName property if your content model associates a name with the content being written, and you want that name associated with the part should the content be embedded at the destination.

The contentSU argument is the content storage unit of a data interchange object. The cloneKind parameter specifies the semantics of the operation and is necessary for cloning. The promiseData will be written into each promised value created by this method in contentSU. The contentShape parameter will be write into the kODPropSuggestedShape property. The partName parameter will be written into the kODPropPartName property.

Note that as shown here, MyWriteToContentSU:

- Is not a SOM method and may throw an exception rather than returning normally
- May only be called once to write to a content storage unit; it should not be used, for example, by CloneInto which may be called multiple times and may need to augment the storage unit on successive calls.

```
void MyWriteToContentSU (Environment *ev,
  ODStorageUnit* contentSU,
```

```
  ODCloneKind cloneKind,
  ODByteArray* promiseData,
  ODShape* contentShape,
  ODIText* partName)
{
  // Begin a clone transaction
  ODDraft* myDraft = fSOMSelf->GetStorageUnit(ev)->GetDraft(ev);
  ODDraftKey draftKey = 0;

  ODVolatile(myDraft);
  ODVolatile(draftKey);

  draftKey = myDraft->BeginClone(ev, contentSU->GetDraft(ev), kODNULL, cloneKind);

  TRY
    contentSU->AddProperty(ev, kODPropContents);

    // Write out the contents, preferred representation first.
    // The type of the best representation is usually
    // the part's own proprietary format.
    // Rather that write the actual content, write a promise.
    // This part will deliver the data in its FulfillPromise method.

    contentSU->SetPromiseValue(ev,
        kMyContentKind,
        0,
        promiseData,
        fSOMThis->fPartWrapper);

    // Promise standard versions of the content, too.
    //   Note that the very same promise data can be used; FulfillPromise can
    //   get the requested type from its ODStorageUnitView* parameter.

    contentSU->SetPromiseValue(ev,
        kODKindOS2Text;
        0,
        promiseData,
        fSOMThis->fPartWrapper);

    // Add or replace the name property (for embedding at the destination)
    if (partName)
      ODSetITextProp(ev, contentSU, kODPropName, kODOS2IText, partName);

    // Add or replace the suggested frame shape
    // (for embedding at the destination)
    if (contentShape)
    {
      ODSUForceFocus(ev, contentSU, kODPropSuggestedFrameShape, kODNULL);
      contentShape->WriteShape(ev, contentSU);
    }

  CATCH_ALL

    myDraft->AbortClone(ev, draftKey);
    RERAISE;

  ENDTRY

  myDraft->EndClone(ev, draftKey);
}
```

Writing an Embedded Frame

When cloning an embedded frame, your part does not add the kODPropContents, kODPropName, or kODPropSuggestedFrameShape properties to the content storage unit. The kODPropContents property is written by the embedded part in its CloneInto method. If the embedded part is cooperative, it can be cloned into the content storage unit such that it will promise its content. The kODPropName property is automatically added by OpenDoc.

Instead, your part typically writes three properties: kODPropContentFrame, kODPropProxyContent, and kODPropCloneKindUsed. Proxy content is optional; if your part does not associate any special information with the embedded frame, you do not need to provide it.

The recipe below must be followed carefully. Create a kODPropContentFrame property in the content storage unit, but DO NOT CLONE THE EMBEDDED FRAME YET. First clone the embedded part, specifying the content storage unit as the destination (this may be the only situation where your part must clone into a specific storage unit). Then clone the embedded frame, and store a WEAK reference to it into a

kODWeakStorageUnitRef value type of the kODPropContentFrame property. The embedded part must be cloned first; if the embedded frame is cloned first, it will force cloning of the embedded part into a storage unit other than the content storage unit. The embedded frame should be weakly referenced so it must be explicitly cloned by the destination.

To enable the embedded part to fulfill a promise for its content, your part must add a kODPropProxyContent property to the content storage unit. This property should contain the clone kind specified in the call to BeginClone.

Although only an embedded frame is selected, the containing part may associate some intrinsic content with that frame, such as the addition of a drop shadow. After the embedded part is cloned into the content storage unit, the containing part can add a kODPropProxyContent property to the content storage unit to contain whatever intrinsic data it chooses. If the destination understands the proxy content, the characteristics of the frame will be preserved.

When a single embedded frame is written, its important that the content storage unit contain a kODPropContentFrame property. The presence of this property tells the destination part that the content came from an embedded frame. According to the OpenDoc Human Interface Guidelines, content cut or copied as an embedded frame should be embedded at the destination, even if the content could be incorporated. Without the kODPropContentFrame property, the destination part cannot know the content came from an embedded frame.

The contentSU argument is the content storage unit of a data interchange object.

The cloneKind parameter specifies the semantics of the operation and is necessary for cloning. The embeddedFrame parameter is the frame being transferred. The optional promiseProxyData will be written into each proxy value created by this method in contentSU.

```
void MyCloneEmbeddedFrameToContentSU (Environment *ev,
  ODStorageUnit* contentSU,
  ODCloneKind cloneKind,
  ODFrame* embeddedFrame,
  ODByteArray* promiseProxyData)
{
  // Begin a clone transaction
  ODDraft* myDraft = fSOMSelf->GetStorageUnit(ev)->GetDraft(ev);
  ODDraftKey draftKey = 0;

  ODVolatile(myDraft);
  ODVolatile(draftKey);

  draftKey = myDraft->BeginClone(ev,
                                 contentSU->GetDraft(ev),
                                 kODNULL,
                                 cloneKind);

  TRY

    // When transferring one frame, add the content frame property
    // now but write the value after the embedded part has been cloned.

    contentSU->AddProperty(ev, kODPropContentFrame);

    // Also add a clone kind used property in case the embedded part promises
    // its content and thus must perform a clone transaction in its
    // FulfillPromise method.

    ODSetULongProp(ev, contentSU, kODPropCloneKindUsed, kODCloneKind, cloneKind);

    // Clone the embedded part into the content storage unit.
    // If the embedded part is savy, it will notice the kODPropContentFrame
    // property and promise its content.

    ODPart* embeddedPart = embeddedFrame->AcquirePart(ev);
    myDraft->Clone(ev,
         draftKey,
         embeddedPart->GetID(ev),
         contentSU->GetID(ev),
         embeddedFrame->GetID(ev));
    ODReleaseObject(ev, embeddedPart);

    // Clone the embedded frame to the data transfer draft;
    // must be done after cloning the part because the
    // embedded frame strongly references the part.

    ODID toFrameID = myDraft->Clone(ev,
         draftKey,
         embeddedFrame->GetID(ev),
         kODNULLID,
         kODNULLID); // scope is not relevant for cloning a frame

    // Weakly reference the frame so it must be explicitly cloned
    //   into the receiving draft if desired

    ODSUForceFocus(ev, contentSU, kODPropContentFrame, kODWeakStorageUnitRef);
    ODStorageUnitRef aSURef = contentSU->GetWeakStorageUnitRef(ev, toFrameID);
```

```
    StorageUnitSetValue(contentSU, ev, sizeof(ODStorageUnitRef), &aSURef);

    // (Optional) Add proxy content describing the embedded frame
    if (promiseData)
    {
      ODSUForceFocus(ev, contentSU, kODPropProxyContents, kODNULL);
      contentSU->SetPromiseValue(ev,
          kMyContentKind,
          0,
          promiseProxyData,
          fSOMThis->fPartWrapper);
    }

  CATCH_ALL

    myDraft->AbortClone(ev, draftKey);
    RERAISE;

  ENDTRY

  myDraft->EndClone(ev, draftKey);
}
```

Implementing CloneInto

Your part may participate in a data interchange operation initiated by some other part. Specifically, your part may be requested to clone itself into a specified storage unit. This storage unit may be on the clipboard, in a drag and drop object, or in a link. Fortunately, it does not matter to your part, as long as your part clones itself correctly.

First, your part should not implement CloneInto by calling its Externalize method and then cloning its storage unit into the specified storage unit. Externalizing your part may require writing to disk; if every part did this in response to CloneInto performance would suffer. Also, your Externalize method does not take the scope frame into account. For example, a display frame of your part may have been cut but not yet removed, and your part should not include this as a display frame when it clones itself into a storage unit.

Your part's CloneInto method should only add and write to the kODPropContents property. OpenDoc will copy the part name and other annotations automatically.

Your part's CloneInto method also should not start a clone transaction by calling BeginClone. One will have already been started by the part initiating the operation.

When CloneInto Should Write a Promise

To enable quick response when the user initiates dragging a frame, all parts should be able to promise their content when their CloneInto method is called. CloneInto should check for the presence of the kODPropContentFrame property. This property will only be present if the part is being cloned into a content storage unit by a containing part. In this situation only, a part can promise its content instead of actually copying data. IN OTHER SITUATIONS YOUR PART MUST WRITE ACTUAL DATA! Your part's CloneInto method may be called when another part is fulfilling a promise, and your part must not promise content in this case.

The promise data written should identify the display frame of this part specified in the scopeFrame parameter to CloneInto, to allow the part's FulfillPromise method to deliver the correct content. Note that CloneInto may be called multiple times, possibly with different scopeFrame parameters.

```
SOM_Scope void  SOMLINK MyPartCloneInto(MyPart *somSelf, Environment *ev,
  ODDraftKey key,
  ODStorageUnit* storageUnit,
  ODFrame* scopeFrame)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","CloneInto");

  // First check for the existence of my content kind; if the value is present,
  //   this part has already been cloned (but respect scopeFrame which could
```

```
  //   include more that was previously cloned; for simple parts this
  //   will not make a difference)
  if (!storageUnit->Exists(ev, kODPropContents, kMyContentKind, 0))
  {
    SOM_TRY

      parent_CloneInto(somSelf, ev, key, storageUnit, scopeFrame);

      // Focus to the contents property, adding if necessary
      ODSUForceFocus(ev, storageUnit, kODPropContents, kODNULL);

      // Optional - Check to see if the content can be promised
      if (storageUnit->Exists(ev, kODPropContentFrame, kODNULL, 0))
      {
        // Placeholders to be replaced by part-specific data
        ODByteArray promiseData = CreateByteArrayStruct(kODNULL,0);

        // Write a promise for each content kind
        storageUnit->SetPromiseValue(ev, kMyContentKind, 0,
                                     &promiseData, somThis->fPartWrapper);

        // Promise other content kinds
        storageUnit->SetPromiseValue(ev, kODKindOS2Text, 0,
                                     &promiseData, somThis->fPartWrapper);

        DisposeByteArrayStruct(promiseData);
      }
      else
      {
        // Placeholders to be replaced by part-specific data
        ODByteArray myContentKindData = CreateByteArrayStruct(kODNULL,0);
        ODByteArray textContentKindData = CreateByteArrayStruct(kODNULL,0);

        // Add content kinds in this part's fidelity order
        ODSUForceFocus(ev, storageUnit, kODPropContents, kMyContentKind);
        storageUnit->SetValue(ev, &myContentKindData);

        ODSUForceFocus(ev, storageUnit, kODPropContents, kODKindOS2Text);
        storageUnit->SetValue(ev, &textContentKindData);

        DisposeByteArrayStruct(myContentKindData);
        DisposeByteArrayStruct(textContentKindData);
      }

    SOM_CATCH_ALL

    SOM_ENDTRY
  }
}
```

Incorporating from a Content Storage Unit

This recipe demonstrates how data may be incorporated from a content storage unit.

To incorporate content, read a value type from the kODPropContents property of the content storage unit. Each storage unit referenced from the value stream should be cloned to the receiving part's draft. The part performing the clone must not internalize objects from cloned storage units until EndClone is called; if the clone has to be aborted for any reason, the storage units will be removed from the draft.

Each frame embedded during incorporation must have their containing frame, link status, and in-limbo fields set correctly. In addition, the previous in-limbo setting of each frame must be remembered for use during undo.

Each frame embedded must be inserted into the receiving part's frame hierarchy by calling the embedded frame's SetContainingFrame method. SetContainingFrame must be called after EndClone since it is illegal to internalize a frame until a cloning transaction successfully completes. SetContainingFrame must be called before the part displayed in the frame is internalized by calling AcquirePart. Internalizing the part causes DisplayFrameConnected to be called. The activation recipe (applied to the embedded part) specifies that the frame should become active if its a root frame, that is, its containing frame is null. The embedded frame will not have a containing frame until one is set by the part performing the incorporation.

If the embedded frame should appear in other display frames of the receiving part, additional embedded frames need to be created. The reader is referred to the Layout recipes for more details on embedding.

Each frame embedded must have its link status set by calling its ChangeLinkStatus method. All parts must set the link status of their

embedded frames, even parts that do not otherwise support linking. Parts that support linking are referred to the Linking recipes for more information on setting link status. Parts that do not support linking should set all embedded frames to kODNotInLink.

This recipe incorporates the part's native content kind, kMyContentKind, from the content storage unit. Unless the user specifies a type to incorporate in the Paste As dialog, the part should incorporate the highest fidelity kind that it supports. This may not be the first kind in the kODPropContents property, nor may it be the part's native representation. The native content kind is incorporated here for simplicity.

The contentSU argument is the content storage unit of the data interchange object to incorporate from. The targetFrame argument identifies the frame being incorporated into and is passed to BeginClone. The cloneKind argument specifies the semantics of the operation and is passed to BeginClone. The embeddedFrameLinkStatus argument is used to set the link status of frames embedded from contentSU.

```
SOM_Scope void  SOMLINK MyPartMyReadFromContentSU(MyPart *somSelf,
                                                  Environment *ev,
  ODStorageUnit* contentSU,
  ODFrame* targetFrame,
  ODCloneKind cloneKind,
  ODLinkStatus embeddedFrameLinkStatus)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","MyReadFromContentSU");

  // Placeholders to be replaced by part-specific data
  ODByteArray data = CreateByteArrayStruct(kODNULL,0);
  ODStorageUnitRef aStrongRef;
  ODID strongClonedID = kODNULLID;
  ODID weakClonedID = kODNULLID;

  // Begin a clone transaction
  ODDraft* myDraft = somSelf->GetStorageUnit(ev)->GetDraft(ev);
  ODDraftKey draftKey = 0;

  ODVolatile(myDraft);
  ODVolatile(draftKey);

  SOM_TRY

    draftKey = myDraft->BeginClone(ev,
                                   contentSU->GetDraft(ev),
                                   targetFrame,
                                   cloneKind);

    TRY

      // For simplicity, this example code incorporates kMyContentKind.
      // A real part should examine the content kinds present
      // in fidelity order and read the best kind it supports.
      contentSU->Focus(ev, kODPropContents, kODPosUndefined,
                       kMyContentKind, 0, kODPosUndefined);

      // For demonstration, read the data into an ODByteArray.
      contentSU->GetValue(ev, contentSU->GetSize(ev), &data);

      // Usually, the data will reference other storage units.
      // These objects need to be cloned into this part's draft.
      // You must wait until EndClone completes without
      // error to turn strongClonedID into a persistent object.
      // If strongClonedID is valid, you may create a storage unit
      // reference to it, but beware that the reference may become
      // invalid after EndClone or AbortClone is called.
      strongClonedID = somSelf->MyCloneStrongReference(ev,
                                                       contentSU,
                                                       aStrongRef,
                                                       draftKey);

      // Your part must distinguish between references that are
      // strong and references that are weak,
      // and clone them accordingly.
      weakClonedID = somSelf->MyCloneWeakReference(ev,
                                                   contentSU,
                                                   aStrongRef,
                                                   draftKey);

    CATCH_ALL

      myDraft->AbortClone(ev, draftKey);
      RERAISE;

    ENDTRY

    myDraft->EndClone(ev, draftKey);
```

```
    // Now incorporate the read data and cloned objects into
    // this part's content
    if (myDraft->IsValidID(ev, strongClonedID))
    {
        // Safe to internalize the cloned object.
        // As an example, the object's
        // storage unit is acquired and released here.
        ODStorageUnit* su = myDraft->AcquireStorageUnit(ev, strongClonedID);
        su->Release(ev);
    }
    else
    {
        // The object was not successfully cloned,
        // As an example, the object's
    }

    // If the incorporated content contains embedded frames,
    // you must ensure certain frame characteristics are set properly.
    // Be sure to call SetContainingFrame before the embedded
    // part is internalized.
    // Also, this part must remember the in-limbo status of each
    // embedded frame in its undo action data to implement
    // undo correctly.
    ODBoolean embeddedFrameWasInLimbo = embeddedFrame->IsInLimbo(ev);
    embeddedFrame->SetInLimbo(ev, kODFalse);
    embeddedFrame->SetContainingFrame(ev, targetFrame);
    embeddedFrame->ChangeLinkStatus(ev, embeddedFrameLinkStatus);

  SOM_CATCH_ALL

  SOM_ENDTRY

  DisposeByteArrayStruct(data);
}
```

Embedding a Content Storage Unit

This example demonstrates how a content storage unit can be embedded as a part. To embed the content storage unit, the part performing the transfer clones the content storage unit into the part's draft. If the content storage unit also contains a kODPropContentFrame property, the storage unit referenced by that property should be explicitly cloned into the part's draft. The frame storage unit must be explicitly cloned because it is only weakly referenced by the content storage unit, and would otherwise not be cloned.

If a frame was provided, the receiving part should call that frame's SetContainingFrame method, passing in the display frame of the active facet. Be sure to call SetContainingFrame before the embedded part is internalized by calling AcquirePart. If no frame was provided, the receiving part should create an embedded frame, using the frame shape specified by the kODPropSuggestedFrameShape property, if present. In either case, for each visible embedded frame, the receiving part should create facets by calling the CreateEmbeddedFacets method of the display frame of the active facet.

The contentSU argument is the content storage unit of the data interchange object to embed. The targetFrame argument identifies the frame being incorporated into and is passed to BeginClone. The cloneKind argument specifies the semantics of the operation and is passed to BeginClone. The embeddedFrameLinkStatus argument is used to set the link status of the embedded frame.

```
SOM_Scope void  SOMLINK MyPartMyEmbedContentSU(MyPart *somSelf, Environment *ev,
  ODStorageUnit* contentSU,
  ODFrame* targetFrame,
  ODCloneKind cloneKind,
  ODLinkStatus embeddedFrameLinkStatus)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","MyEmbedContentSU");

  ODDraft* contentSUDraft = contentSU->GetDraft(ev);
  ODDraft* myDraft = somSelf->GetStorageUnit(ev)->GetDraft(ev);

  ODDraftKey draftKey = 0;
  ODID embeddedFrameID = kODNULLID;
  ODID embeddedPartID = kODNULLID;

  ODPart* embeddedPart = kODNULL;
  ODFrame* embeddedFrame = kODNULL;

  // This part must remember the in-limbo status of the embedded frame in its
```

```
//   undo action data to implement undo correctly.
ODBoolean embeddedFrameWasInLimbo = kODTrue;

// Placeholders to be replaced by part-specific data
ODByteArray proxyData = CreateByteArrayStruct(kODNULL,0);

ODVolatile(contentSUDraft);
ODVolatile(draftKey);

SOM_TRY

  // Begin a clone transaction into this draft.
  draftKey = contentSUDraft->BeginClone(ev, myDraft, targetFrame, cloneKind);

  TRY

    // Clone the content storage unit
    //   The scopeID kODNULLID indicates the clone is not restricted to a
    //   particular frame context.
    embeddedPartID = contentSUDraft->Clone(ev, draftKey, contentSU->GetID(ev),
                                           kODNULLID, kODNULLID);

    // If a content frame is present, clone it, too
    if (ODSUExistsThenFocus(ev,
                            contentSU,
                            kODPropContentFrame,
                            kODWeakStorageUnitRef))
    {
      ODStorageUnitRef aSURef;
      StorageUnitGetValue(contentSU, ev, sizeof(ODStorageUnitRef), &aSURef);
      embeddedFrameID = somSelf->CloneStrongReference(ev,
                                                      contentSU,
                                                      aSURef,
                                                      draftKey);
    }

    // Read proxy content if present.
    if (ODSUExistsThenFocus(ev,
                            contentSU,
                            kODPropProxyContents,
                            kMyContentKind))
    {
      // For demonstration, read the data into an ODByteArray.
      // Note that this may contain storage unit references,
      // such as to an ODLink or ODLinkSource storage unit,
      // so cloning may be necessary here.
      contentSU->GetValue(ev, contentSU->GetSize(ev), &proxyData);
    }

  CATCH_ALL

    contentSUDraft->AbortClone(ev, draftKey);
    RERAISE;

  ENDTRY

  contentSUDraft->EndClone(ev, draftKey);

  if (myDraft->IsValidID(ev, embeddedFrameID))
  {
    embeddedFrame = myDraft->AcquireFrame(ev, embeddedFrameID);
    embeddedFrameWasInLimbo = embeddedFrame->IsInLimbo(ev);
    embeddedFrame->SetInLimbo(ev, kODFalse);
    embeddedFrame->SetDragging(ev, kODFalse); // In case the frame was dropped
    embeddedFrame->SetContainingFrame(ev, targetFrame);
  }
  else
  {
    // Create a new embedded frame for newPart,
    // using the suggested frame shape if present.
    TempODShape newShape = NULL;

    if (ODSUExistsThenFocus(ev,
                            contentSU,
                            kODPropSuggestedFrameShape,
                            kODNULL))
    {
      newShape = targetFrame->CreateShape(ev);
      newShape->ReadShape(ev, contentSU);
    }

    // Acquire the new embedded part
    TempODPart embeddedPart = myDraft->AcquirePart(ev, embeddedPartID);
```

```
      // Use my method to create an embedded frame of the desired shape
      embeddedFrame = somSelf->MyMakeEmbeddedFrame(ev,
                                                   targetFrame,
                                                   newShape,
                                                   embeddedPart);
    }

    // Set the link status of the embedded frame
    embeddedFrame->ChangeLinkStatus(ev, embeddedFrameLinkStatus);

    // Add the embedded frame and proxy content to this part's content
    //...

    // Now add facets to display the newFrame
    //...

  SOM_CATCH_ALL

  SOM_ENDTRY
}
```

Creating an Embedded Frame

This example demonstrates how a part could make a new frame for a part embedded from a content storage unit. This is mostly a wrapper around ODDraft::CreateFrame.

The containingFrame argument is the frame to contain the new embedded frame. The frameShape argument is the desired shape or null to use a default shape. The embeddedPart argument is the part to be displayed by the embedded frame.

```
SOM_Scope ODFrame* SOMLINK MyPartMyMakeEmbeddedFrame(MyPart *somSelf,
                                                     Environment *ev,
  ODFrame* containingFrame,
  ODShape* frameShape,
  ODPart* embeddedPart)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","MyMakeEmbeddedFrame");

  const ODBoolean notRoot = kODFalse;
  const ODBoolean notOverlaid = kODFalse;
  ODCanvas* nullCanvas = (ODCanvas*) kODNULL;

  ODFrame* embeddedFrame = kODNULL;
  TempODShape newShape = kODNULL;  // Automatically released

  SOM_TRY

    if (frameShape)
    {
      newShape = frameShape;
      newShape->Acquire(ev);
    }
    else
    {
      // No suggested frame shape, so use a default size
      ODRect rect;
      rect.SetInt(0,0,80,80);
      newShape = containingFrame->CreateShape(ev);
      newShape->SetRectangle(ev, &rect);
    }

    embeddedFrame = somSelf->GetStorageUnit(ev)->GetDraft(ev)->CreateFrame(ev,
                        kODFrameObject,
                        containingFrame,
                        newShape,
                        nullCanvas,
                        embeddedPart,
                        _fSession->Tokenize(ev, kODViewAsFrame),
                        _fSession->Tokenize(ev, kODPresDefault),
                        notRoot,
```

```
                        notOverlaid);
  SOM_CATCH_ALL

  SOM_ENDTRY

  return embeddedFrame;
}
```

Paste As Dialog

Two data interchange objects, the clipboard and the drag and drop object, have a ShowPasteAsDialog method that present a dialog allowing the user to specify options on paste or drop.

The ODPasteAsMergeSetting parameter to ShowPasteAsDialog specifies whether **Merge with Contents** or **Embed as** is initially chosen, and whether the other choice is available. This parameter allows the Paste As dialog to default to the same behavior your part uses on Paste. Specify this parameter using one of these ODPasteAsMergeSetting values:

kODPasteAsMerge	**Merge with Contents** is initially selected; **Embed as** allowed. Specify this constant if your part can incorporate one of the kinds in the content storage unit, and your part supports embedding.
kODPasteAsEmbed	**Embed as** is initially selected; **Merge with Contents** is allowed. Specify this constant if your part can incorporate one of the kinds in the content storage unit, but your part would prefer to embed it. This is the case if incorporation would result in a loss of fidelity, or if the content storage unit was created by copying a single embedded frame (that is, the content storage unit contains a kODPropContentFrame property).
kODPasteAsMergeOnly	**Embed as** is disabled. Specify this constant if your part does not support embedding, and your part can incorporate one of the kinds in the content storage unit, possibly translated.
kODPasteAsEmbedOnly	**Merge with Contents** is disabled. Specify this constant if your part supports embedding, and no kind in the content storage unit can be incorporated, even after translation.

The ShowPasteAsDialog method returns its results in the ODPasteAsResult parameter. If ShowPasteAsDialog returns kODTrue, your part is responsible for disposing the selectedKind, translateKind, and editor fields of this structure. ShowPasteAsDialog will set any unspecified fields to nil.

Embedding a Part via the Paste As Dialog

When a part is embedded via the Paste As dialog, the user can specify a particular content kind or request translation to another kind. In addition, the user may specify the editor to bind to the embedded part. The caller of ShowPasteAsDialog must perform a few simple steps to ensure these choices are followed. This example uses the ShowPasteAsDialog method of the clipboard object.

```
ODPasteAsResult pasteAsResult;

// Parameters except pasteAsResult are omitted
if (clipboard->ShowPasteAsDialog(..., &pasteAsResult))
{
  if (pasteAsResult.mergeSetting == kODFalse)
  {
    // Clone the root storage unit from the clipboard as demonstrated
    //   in the general embedding recipe above
    key = clipDraft->BeginClone(...);
    newPartID = clipDraft->Clone(...)
    ...
    clipDraft->EndClone(...);
```

```
    // Acquire the newly embedded storage unit
    ODStorageUnit* newSU = myDraft->AcquireStorageUnit(ev, newPartID);

    if (pasteAsResult.translateKind != kODNULL)
    {
      // Translation was requested from translateKind to selectedKind
      // Add selectedKind and create the to storage unit view
      ODSUForceFocus(ev, newSU, kODPropContents, pasteAsResult.selectedKind);
      ODStorageUnitView* toView = newSU->CreateView(ev);

      // Create the from storage unit view
      newSU->Focus(ev, kODPropContents, kODPosUndefined,
                   pasteAsResult.translateKind, 0, kODPosUndefined);
      ODStorageUnitView* fromView = newSU->CreateView(ev);

      // Get the translation object and perform the translation
      ODTranslation* translation = mySession->GetTranslation(ev);
      ODTranslateResult translatResult = translation->
                                         TranslateView(ev, fromView, toView);

      // Check for successful translation
      ...

      // Clean up after translation
      toView->Release(ev);
      fromView->Release(ev);
    }

    // Set the preferred kind
    ODSetISOStrProp(ev, contentSU, kODPropPreferredKind,
                    kODISOStr, pasteAsResult.selectedKind);

    // Set the preferred editor
    if (pasteAsResult.editor != kODNoEditor)
    {
      ODSetISOStrProp(ev, contentSU, kODPropPreferredEditor,
                      kODEditor, pasteAsResult.editor);
    }

    // Release the storage unit
    newSU->Release(ev);
  }
}

// Complete embedding of the part
...

// When finished with pasteAsResult, be sure to dispose of the
//   selectedKind, translateKind, and editor fields.
if (pasteAsResult.selectedKind != kODNULL)
  DisposePtr(pasteAsResult.selectedKind)

if (pasteAsResult.translateKind != kODNULL)
  DisposePtr(pasteAsResult.translateKind)

if (pasteAsResult.editor != kODNULL)
  DisposePtr(pasteAsResult.editor)
```

Drag and Drop

The Drag and Drop protocol provides a direct manipulation alternative to the clipboard for copying and moving data between parts in a document, between documents and between documents and the desktop.

This document contains a number of recipes for transferring data through the OpenDoc drag and drop mechanism. Refer to the Class Documentation for a description of individual methods.

The examples use exception handling in the form of TRY...CATCH_ALL...ENDTRY blocks to catch exceptions, and assumes that errors returned by SOM methods are automatically thrown upon return.

Note: Code appearing in these recipes has not been tested, and may contain errors. They serve as examples of how to use the OpenDoc functions to accomplish specific objectives in particular circumstances. Actual part code usually needs to draw from several recipes to

handle the variety of conditions that occur in a real application.

The drag and drop focus (or the lack of it)
There is no drag and drop focus because only one part can be initiating a drop at a time and the drag is synchronous.

Undo of a drag and drop
A copy or move using drag and drop should be undoable. An undo transaction should be started by the part initiating the drag; the part receiving the drop may add undo actions to the transaction. The transaction should be ended after the drop completed by the part initiating the drag. See the Undo recipe for details.

Embedded frames and their in-limbo flag
When an embedded frame is dragged or dropped, parts must set the in-limbo flag of the frame correctly. Parts are also responsible for setting the in-limbo flag when a drag and drop operation is undone, redone, or committed via their DisposeActionState method. The code examples here demonstrate the proper manipulation of this flag during the drag and drop, but do not cover how your part must support the in-limbo flag in your UndoAction, RedoAction, or DisposeActionState methods. The complete recipes for setting this flag, and for the actions to take in DisposeActionState, are discussed in the OpenDoc Programmer's Guide.

Preparing for a Drop

A part which can receive a drop should call SetDroppable on all its display frames which can accept a drop. This can be done during ODPart::DisplayFrameAdded or ODPart::DisplayFrameConnected.

```
SOM_Scope void  SOMLINK MyPartDisplayFrameAdded(AppleTest_Container *somSelf,
                                                Environment *ev,
                                                ODFrame* frame)
{
 ...
 frame->SetDroppable(ev, kODTrue);
 ...
}
```

If your part does not call SetDroppable on a display frame, the part will not receive any DragEnter, DragWithin, DragLeave and Drop from any facet on that frame.

If your part no longer wishes to receive drops from a particular frame, it can call ODFrame::SetDroppable with kODFalse.

Detecting a Drag (Handling Mouse-Down Events)

Note that a drag can be initiated by any part even when the part does not have any display frame that is set to receive drops.

A drag is initiated by a WM_BEGINDRAT received by your HandleEvent method. (For more information on handling events, see the Basic Event Handling recipe). When handling a WM_BEGINDRAG, check to see if the coordinates of the mouse pointer in the facet are within an item that can be dragged, such as a content item, a text selection, or a selected or bundled embedded frame.

Initiating a Drag

Once a part knows that a drag can be initiated, the source part should call ODSession's GetDragAndDrop to get the ODDragAndDrop object. Then it should call ODDragAndDrop's Clear and ODDragAndDrop's GetContentStorageUnit to get the ODStorageUnit to where data of the dragged object(s) is copied. Then it can initiate drag by calling ODDragAndDrop's StartDrag. An image for dragging feedback is provided by the source part.

Drag outline:
The imageType is DRAGIMAGE and the image data can be either a bit map or a polygon outlining the drag begin. (For more information, see the description of the DRAGIMAGE data type in the *OpenDoc Programming Reference* .) It is the part's responsibility to fill out the DRAGIMAGE structure.

Dragging a frame:
In general, it is a bad idea to drag a frame into itself. Therefore, if you are dragging a frame or a set of frames, you should call

ODFrame::SetDragging using kODTrue. This will tell OpenDoc drag and drop that these frames are not supposed to accept this drop.

Mouse-down offset:
The source part should write out the mouse-down offset from the bottom-left corner of the selection. This enables the destination part to place the selection at the correct offset from the mouse up position when it is dropped.

Clone kind:
If the source part needs to clone its content (whether it is intrinsic or embedded parts), it should use kODCloneCut. kODCloneCopy is only used when the content cannot be moved. If kODCloneCopy is used, kODDropMove will never be returned as the return result of ODDragAndDrop::StartDrag.

Recipe:

```
// Get the ODDragAndDrop object from the session.

ODDragAndDrop* dragAndDrop =
     somSelf->GetStorageUnit(ev)->GetSession(ev)->GetDragAndDrop(ev);

// Reinitialize the ODDragAndDrop object.
dragAndDrop->Clear(ev);

// Get the Storage Unit where data for dragged objects are going to be written.
ODStorageUnit* storageUnit = dragAndDrop->GetContentStorageUnit(ev);

// Write out the data.
// Refer to the following sections in Clipboard recipes:
// Annotating the clipboard with a frame shape
// Writing a frame shape to the clipboard
// Always use BeginClone and EndClone
// Putting intrinsic content on the clipboard
// Copying content to the clipboard
// Putting a single embedded frame on the clipboard.
...

// If you are dragging a frame or a set of frames,
// you should notify OpenDoc drag and drop
// that this frame is being dragged.
frame->SetDragging(ev, kODTrue);

// Anticipate a move by marking all dragged frames as in-limbo
frame->SetInLimbo(ev, kODTrue);

// Write out the mouse down offset.
storageUnit->AddProperty(ev, kODPropMouseDownOffset);
storageUnit->AddValue(ev, kODPoint);

// Calculate offset
...

// Write out the mouse down offset.
storageUnit->SetValue(ev, &ba);

DRAGIMAGE dimg;

// Initiate the DRAGIMAGE structure.
...

// Create byte arrays for drag image and refCon (which is the event record)

ODByteArray dragImageBA;
dragRgnBA._length = sizeof(DRAGIMAGE);
dragRgnBA._maximum = sizeof(DRAGIMAGE);
dragRgnBA._buffer = &dimg;

ODByteArray eventBA;
eventBA._length = sizeof(ODEventData*);
eventBA._maximum = sizeof(ODEventData*);
eventBA._buffer = &event;        // event is of type ODEventData*.

// Initiate the drag
ODPart* destPart;

dropResult =
 dragAndDrop->StartDrag(sourceFrame, 0, &drgImagBA, &destPart,
                        &eventBA);

// If you have set any frame to not accept any drop, unset it.
frame->SetDragging(ev, kODFalse);
```

```
// If the drag was not a move, dragged frames are no longer in limbo
if (dropResult != kODDropMove)
{
  frame->SetInLimbo(ev, kODFalse);
}

if (dropResult == kODDropMove)
{
  // If a frame or set of frames was moved, old facets must be removed.
  // The source part must take into account that a moved frame may
  // you should notify OpenDoc drag and drop in another part,
  // and may have new facets in its new containing frame.
  // The source part cannot use the moved frame's facet iterator.
  // It must instead use the containing facet's iterator to identify
  // the facets to be deleted.
  ...
}

if (destPart != kODNULL)
  destPart->Release(ev);
```

Tracking a Drag

ODPart's DragEnter is called when the mouse enters a facet. The part should examine the available data types of the dragged items using the ODDragItemIterator passed in. If the part can handle a drop of the dragged object, it should provide the appropriate feedback (e.g., adorning the droppable frame, changing the cursor). If the destination part cannot handle the data types, nothing should be done.

If there is more than one drag item, the Part should make sure that it can accept all the drag items. If there is one or more drag item that the Part cannot accept, it should return kODFalse from DragEnter and do no visual feedback.

The following code fragment shows a simple example where a part which can only accepts one kind (kMyKind) determines whether the current drag can be accepted.

```
ODBoolean CanAcceptThisDrop(Environment* ev, ODDragItemIterator* dragInfo)
{
 // assuming that we can accept all the drag items
 ODBoolean canAccept = kODTrue;
 for (ODStorageUnit* dragSU = dragInfo->First(ev);
   (dragInfo->IsNotComplete(ev) && (canAccept == kODTrue));
   dragSU = dragInfo->Next(ev))
 {
  if (dragSU->GetSession(ev)->GetDragAndDroop(ev)->CanIncorporate(ev, dragSU,
                                                    kMyKind))==kODFalse)
   canAccept = kODFalse;
 }
 return canAccept;
}
```

However, checking the kinds is not enough to provide the full feedback according to the Drag Manager guidelines. The guidelines dictate that no highlighting should be shown on the frame from which the drag is initiated until the drag has left and returned to the frame.

In order to help part developers implement this, OpenDoc drag and drop provides two Drag Attributes:

kODDragIsInSourceFrame shows that the drag still has not left the source frame. kODDragIsInSourcePart shows that the drag still has not left the source part.

```
ODDragResult MyPartDragEnter(MyPart* somSelf,
      Environment* ev,
      ODDragItemIterator* dragInfo,
      ODFacet* facet,
      ODPoint where)
{

 // Get the drag and drop object associated with this drag.
 // Note that the session from dropSU may not be the same as the session of the
 // target part.
 ODDragAndDrop* dad = dropSU->GetSession(ev)->GetDragAndDrop(ev);
 ODULong dragAttributes = dad->GetDragAttributes(ev);

 // Determine whether we can handle this drag.
```

```
 ODBoolean canAccept = CanAcceptThisDrag(ev, dragInfo);

 // If we can accept the data and the drag has left the source frame,
 // do some adornment.
 // Here we are going to put up drag hilite on the facet.
 // We're calling our part's InvertDragHilite method (shown below).

 if ((canAccept == kODTrue) && !(dragAttributes & kODDragIsInSourceFrame)) {
  somSelf->InvertDratHilite(ev, facet);

 // Return ODDragResult to show whether a Drop can happen in this facet.
 return canAccept;
}

void MyPartInvertDragHilite(MyPart* somSelf, Environment * ev, ODFacet* facet)
{
   // this method toggles highlighting for the facet

   TRY

   HPS hps;

   CFocusWindow f(ev, facet, (ODShape*)kODNULL, &hps, (HWND*)kODNULL,
                  CFocusWindow::DragPS);

   ODTempPolygon poly;
   POLYGON polygon;
   ODContour *pContour;
   int i;
   TempODShape clipShape = facet->AcquireAggregateClipShape(ev, kODNULL);
   clipShape->CopyPolygon(ev,&poly);     // [137664]

   GpiSetMix(hps, FM_INVERT);
   GpiBeginPath(hps, 1);
   for (pContour = poly.FirstContour(), i = 0;
        i < poly.GetNContours();
        pContour = pContour->NextContour(), i++)
   {
      pContour->AsPOLYGON(polygon);
      GpiMove(hps, &polygon.aPointl[polygon.ulPoints-1]);
      GpiPolyLine(hps, polygon.ulPoints, polygon.aPointl);
      SOMFree(polygon.aPointl);
   }
   GpiEndPath(hps);
   GpiSetLineWidthGeom(hps, 8);
   GpiStrokePath(hps, 1, 0);

   CATCH_ALL
   ENDTRY
}
```

Within a part's facet:
ODPart's DragWithin is called continuously when the mouse is still in the facet. This allows the part to do any processing desired. One good example is when the frame has several hot spots where objects can be dropped. If the mouse is not over these hot spots, the cursor may need to be changed to reflect that the no dropping can be done there even though it is still in a droppable frame. Again, a ODDragItemIterator is passed in so that the part can examine the availabe data types of the dragged objects.

ODPart's DragWithin also provides a chance for the part to examine the state of the machine. For example, some part may want to find out whether the modifier keys are down or not.

Leaving a part's facet:
ODPart's DragLeave is called when the mouse leaves a droppable frame. This allows the part to clean up after a drag within it (e.g., removing adornment on the frame, changing the cursor back to its original form).

Receiving a Drop

If the mouse is released within a facet, ODPart's Drop is called on the part which owns the facet. The part can then figure out whether it can receive the dragged object using the ODDragItemIterator passed in.

OpenDoc User Interface Guidelines defines guidelines for the destination part to decide whether a drop should be a move or a copy. OpenDoc drag and drop provides this information to the part through the OpenDoc Drag Attributes . The part can get the OpenDoc Drag Attributes through the ODDragAndDrop object. When ODDragAndDrop::GetDragAttributes is called, a ODULong is returned. The value of the OpenDoc Drag Attributes reflects important information about the Drop:

kODDropIsInSourceFrame	The drop happens in the frame where the drag is initiated.
kODDropIsInSourcePart	The drop happens in the part when the drag is initiated.
kODDropIsMove	The drop is a move.
kODDropIsCopy	The drop is a copy.
kODDropIsLink	The drop is a link (OS/2 only)
kODDropIsPasteAs	The destination part should put up the Paste As dialog box and enable the user to take further actions.

The destination part should look for kODPropMouseDownOffset to position the dropped data according to the original mouse-down offset.

```
ODDropResult MyPartDrop(MyPart* somSelf,
              Environment* ev,
              ODDragItemIterator *dropInfo,
              ODFacet* facet,
              ODPoint where)
{
 ...

 // Determine whether we can handle this drag.
 ODBoolean canAccept = CanAcceptThisDrag(ev, dragInfo);

 // Get the OpenDoc Drop Attirbutes for this drop
 ODDragAndDrop* dad = somSelf->
                      GetStorageUnit(ev)->
                      GetSession(ev)->
                      GetDragAndDrop(ev);
 ODULong dragAttributes = dad->GetDragAttributes(ev);

 ODDropResult dropResult = kODDropFail;

 if(dragAttributes & kODDropIsMove)
        dropResult = kODDropMove;
 else if(dragAttributes & kODDropIsCopy)
        dropResult = kODDropCopy;

 ODStorageUnitView   *dropView;
 ODStorageUnit       *dropSU, *renderedSU;

 for (dropSU = dropInfo->First(ev);
              dropInfo->IsNotComplete(ev);
              dropSU = dropInfo->Next(ev))
 {
   // first thing is to check for dragitem value, can't do anything
   // more if it is not present...
   if (!dropSU->Exists(ev, kODPropContents, (ODValueType)kODDragitem, 0))
   {
      dropResult = kODDropFail;
      break;
   }

   dropSU->Focus(ev, kODPropContents, kODPosUndefined,
                        (ODValueType)kODDragitem,   0, kODPosUndefined);

   // create a view of our focused value for the drag&drop object
   ODStorageUnitView *dropView = dropSU->CreateView(ev);

   // ask the drag&drop object to render the data
   if(!dropSU->GetSession(ev)->GetDragAndDrop(ev)->GetDataFromDragManager(ev,
                                      dropView,
                                      &renderedSU))
   {
      dropResult = kODDropFail;
      break;
   }
   if(dragAttributes & kODDropIsPasteAs) {
    // Refer to the following clipboard recipe:
    // Embedding a part via the Paste As Dialog
    ...
   }
   else {
    // Refer to the following clipboard recipes:
    // Incorporating content from the clipboard
    // Embedding a part from the clipboard
    ...
   }
```

```
    // For each frame embedded during the drop,
    // remember its current in-limbo status
    // before setting the status to false (not in-limbo)
    wasInLimbo = frame->IsInLimbo(ev); // for use during undo
    frame->SetInLimbo(ev, kODFalse);
  }
  return dropResult;
}
```

Move, Copy, Paste As:
The destination part can also override a move and make the drop into a copy. However, changing a copy to a move is not allowed.

Returning the drop result:
After the destination part has responded to the drop, it needs to notify the source part whether it has accepted the drop as a move or a copy. The following are the predefined values of ODDropResult:

kODDropFail	The drop has failed.
kODDropCopy	The drop is a copy.
kODDropMove	The drop is a move.
kODDropUnfinished	The drop is not finished. This value should never be returned by the destination part. It is only use by OpenDoc when DragAndDrop::StartDrag returns immediately in the case of an asynchronous drag.

This result is returned to the source part (via the system) whether the drop is accepted and what action the source part should take.

Incorporating Data from a Non-OpenDoc Document

When a Part's Drop method is called, it is given an ODDragItemIterator. ODDragItemIterator allows the user to access a collection of Drag Items. As described above, it is the receiver's (or the destination part's) responsibility to iterate through all the Drag Items to find out whether it can receive the Drop.

If the Drop comes from an OpenDoc part, there is only one Drag Item in the collection. If the Drop comes from a non-OpenDoc application (for example, the Workplace Shell), there may be one or more Drag Item. If there is more than one Drag Item, the destination part can only accept the drop if it can accept all the drag items.

If the drag was initiated by Workplace Shell and the dropped object is a file the storage unit rendered by GetDataFromDragManager will contain the following values for its kODPropContents property:

The selected kind	The part kind the selected RMF was mapped to; no value is set for this kind, the actual data has to be read from the dropped file.
kODFileType	The name of the dropped file.
kODFileTypeEA	Any extended attributes that are associated with the file.

Embedding Data from a Non-OpenDoc Document

Even when the destination part cannot incorporate the data, there may be other parts on the system which can. Following the OpenDoc model, the destination part can then embed the non-OpenDoc file and let another Part Editor handle the data.

From the destination's point of view, the recipe for embedding data from a non-OpenDoc document is no different from embedding an OpenDoc part. It first clones the Content Storage Unit and then it calls GetPart using the cloned Storage Unit. The OpenDoc Binding mechanism will use the corresponding Part Editor to manipulate the content of the cloned Storage Unit.

Here's what the destination part should do in its Drop method:

```
...

ODDraft* dadDraft = dropSU->GetDraft(ev);
ODDraft* myDraft = somSelf->GetStorageUnit(ev)->GetDraft(ev);
```

```
ODDraftKey key = 0;
ODID newPartID = kODNULLID;
ODPart* newPart = kODNULL;

ODVolatile(key);
ODVolatile(newPartID);

SOM_TRY

 // Begin a transfer from the drag and drop container into this draft.
 key = dadDraft->BeginClone(ev, myDraft, facet->GetFrame(ev), kODClonePaste);

 // Clone the Storage Unit (which contains information on the
 // non-OpenDoc document) from the drag and drop container.
 //   Since we are cloning only a Storage Unit, there is no scoping. 0
 //   is passed in to indicate that.
 newPartID = dadDraft->Clone(ev, key, dropSU->GetID(ev), 0, 0);

 dadDraft->EndClone(ev, key);

SOM_CATCH_ALL

 if (key != 0)
  dadDraft->AbortClone(ev, key);

 newPartID = kODNULLID;

SOM_ENDTRY

// Get the new embedded part
 if (newPartID == kODNULLID) {
  // cleanup and return error
 }
 else {
  newPart = myDraft->AcquirePart(ev, newPartID);
 }

...
```

Here's what the newly created part should do in its InitPartFromStorage method:

```
CHAR szFile[CCHMAXPATH];
// get the dropped file name
if (myStorageUnit->Exists(ev, kODPropContents, (ODValueType)kODFileType, 0))
{
   myStorageUnit->Focus(ev,
                        kODPropContents,
                        kODPosUndefined,
                        (ODValueType)kODFileType,
                        0,
                        kODPosUndefined);
   ULONG dataSize = myStorageUnit->GetSize(ev);
   StorageUnitGetValue(myStorageUnit, ev, dataSize, (ODValue) szFile);

   // check for the existance of a non-OPENDOC kind the part supports;
   // that should be the type of data in the file;
   // this has to be done for all such kinds, until a match is found
   if (myStorageUnit->Exists(ev, kODPropContents, myOS2Kind, 0)) {
     // open the file and read in the data
     ....
   }
   // another option would be to determine the file type by reading
   // the kODFileTypeEA value

   // remove any undesired values and add the primary value kind
   // to myStorageUnit
}
```

Promises

The basic mechanisms for transferring data between Parts are the Clipboard and drag and drop. If the data to be exported is very big, the part

may choose to export the data as a promise. When the data is retrieved by the destination part, the source part will be requested to fulfill the promise it put out.

The format of a Promise is determined by the Part. The only restriction is that a Promise must be able to be written to a Storage Unit Value.

Putting Out a Promise

The following code fragment shows how a Part can put out a Promise in response to a mouse down event and a Drag is going to be initiated. (This recipe can also be used to put out a promise to the Clipboard).

```
ODBoolean MyPartHandleDragging(MyPart* somSelf,
                               Environment* ev,
                               ODFacet* facet,
                               ODPoint where,
                               ODEventData event)
{

 ...

 // Get the ODDragAndDrop object from the session.
 ODDragAndDrop* dragAndDrop =
     this->GetStorageUnit(ev)->GetSession(ev)->GetDragAndDrop(ev);

 // Reinitialize the ODDragAndDrop object.
 dragAndDrop->Clear(ev);

 // Get the Storage Unit where data for dragged objects is going to be
 // written.
 ODStorageUnit* storageUnit = dragAndDrop->GetContentStorageUnit(ev);

 // Focus the Storage Unit to the Property.
 storageUnit->Focus(ev,
                    kODPropContents,
                    kODPosUndefined,
                    kODNULL,
                    0,
                    kODPosUndefined);

 // Prepare the promise. This section is Part specific.
 MyPromise promise = ...

 // Put the promise data in an ODByteArray.
 ODByteArray ba;
 ...

 // Put out the promise.
 storageUnit->SetPromiseValue(ev,
       kODKinDOS2Text,       // type of promised data
       0,                    // offset
       &ba,              // promise
       fPartWrapper);        // source part

 // Initiate the Dragging
 // Follow the drag and drop recipe.
 .........

}
```

Getting Data from a Value with a Promise

When the destination part retrieves the data (using ODStorageUnit::GetValue or ODStorageUnit::GetSize), the source part will be called to fulfill the promise. The destination part does not even know that the fulfilment of a promise is taking place.

The following is what the code for the destination part might look like. The same code is used whether the Value contains a promise or not.

```
ODDropResult YourPartDrop(YourPart* somSelf,
                          Environment* ev,
                          ODDragItemIterator* dropInfo,
                          ODFacet* facet,
                          ODPoint where)

{

 ODDropResult    dropResult = kODDropFail;
 ODStorageUnit  *dropSU;
 ODBoolean       notDone = kODTrue;
 ODStorageUnit  *renderedSU;

 // Iterate through the types and find a desired type
 for (dropSU = dropInfo->First(ev);
      dropInfo->IsNotComplete(ev) && notDone;
      dropSU = dropInfo->Next(ev))
 {
  if (!dropSU->Exists(ev, kODPropContents, (ODValueType)kODDragitem, 0))
  {
     dropResult = kODDropFail;
     break;
  }

  dropSU->Focus(ev, kODPropContents, kODPosUndefined,
                        (ODValueType)kODDragitem,   0, kODPosUndefined);

  // create a view of our focused value for the drag&drop object
  ODStorageUnitView *dropView = dropSU->CreateView(ev);

  // ask the drag&drop object to render the data
  if(!dropSU->GetSession(ev)->GetDragAndDrop(ev)->GetDataFromDragManager(ev,
                                      dropView,
                                      &renderedSU))
  {
     dropResult = kODDropFail;
     break;
  }
  // If the desired type exists, import the data.
  if (dropSU->Exists(ev, kODPropContents, kAppleText, 0) == kODTrue) {

  // Focus to the property and value containing the desired data.
  renderedSU->Focus(ev,
                    kODPropContents,
                    kODPosUndefined,
                    kODKindOS2Text,
                    0,
                    kODPosUndefined);

  // Get the size of the data.
  // **** Note that the promise is being fulfilled here. But this Part
  // **** does not even realize that.

  ODULong size = renderedSU->GetSize(ev);

  // Create an ODByteArray to hold the returned data.
  // The fields will be filled out by GetValue.
  ODByteArray ba;

  renderedSU->GetValue(ev, size, &ba);

  // Put the data into the part's content.
  ...

  // Dispose the byte array buffer.
  SOMFree(ba._buffer);

  // Stop the iteration.
  notDone = kODFalse;

  // Set up drop result appropriately.
  dropResult = kODDropCopy;

  }
 }

 // Return the appropriate result code.
 return dropResult;

}
```

Fulfilling a Promise

The following code fragment is an example of what the source part might do in response to a request to fulfill a promise:

```
void MyPartFulfillPromise(MyPart* somSelf,
                          Environment* ev,
                          ODStorageUnitView* promiseSUView)

{
 // Do a check first to see whether it is the right size.
 ODULong size = promiseSUView->GetSize(ev);

 if (size == sizeof(MyPromise)) {

  // Get the Promise.
  // Either GetValue or GetPromiseValue can be used here.
  // OpenDoc ensures that if GetValue is used,
  // remember its current in-limbo status
  // (i.e., GetValue -> FulfillPromise -> GetValue -> FulFillPromise).

  // Set up the byte array for promise.
  MyPromise  promise;
  ODByeArray promiseByteArray;
  promiseByteArray._length = promisedDataSize;
  promiseByteArray._maximum = promisedDataSize;
  promiseByteArray._buffer = &promise;

  promiseSUView->GetValue(ev, (ODValue) &promiseByteArray);

  // Get the Promised Data using the Promise.
  // This section is part-specific.

  ODPtr promisedData = ......(promise);

  // Set up the byte array for promisedData.
  ODByeArray promisedDataByteArray;
  promisedDataByteArray._length = promisedDataSize;
  promisedDataByteArray._maximum = promisedDataSize;
  promisedDataByteArray._buffer = promisedData;

  // Write the promised data back to the Value.
  // One can use use the promiseSUView to get the Storage Unit and write out
  // other Properties and Values other than the one refered to by promiseSUView.
  promiseSUView->SetValue(ev, (ODValue) &promisedDataByteArray);

  // Cleanup - only need to dispose the promisedData.
  // The byte array is a stack variable.
  SOMFree(promisedData);
  }
 else {

  // Simply delete the promise.
  promiseSUView->DeleteValue(ev, size);

  // The destination will simply see a Value with no content.
  // It is the responsibility of the destination part to handle invalid values.
  // This is no different from the general case without promises.

 }
}
```

Forcing Promise Fulfillment

There are times when your part might need to force the fulfillment of a promise it has written, before another part actually tries to read the data. An example of this might be when your part takes some action that would prevent it from fulfilling the promise in the future. The simplest way for a part to force a promise to be fulfilled is to focus on the promised property/value and call the storage unit routine GetSize.

Drag and Drop - Promising Non-OpenDoc File

The Drag and Drop mechanism allows the source part to create a non-OpenDoc file when the destination is the Finder or other applications which can accept a file.

Initiating a Drag

If the source part puts a promise value of "DRM_OS2FILE, DRF_UNKNOWN" type in the contents property of the content storage unit of the ODDragAndDrop object, the Drag and Drop mechanism would ask the source part to fulfill the promise when a drop happens in the Workplace Shell or other applications which can accept a file.

The following is the code fragment for creating such a promise. This piece of code is called right before ODDragAndDrop::StartDrag.

```
ODByteArray         thePromise;
dadsu->Focus(ev, kODPropContents, kODPosUndefined, kODNULL, 0, kODPosUndefined);
dadsu->SetPromiseValue(ev, "DRM_OS2FILE, DRM_UNKNOWN", 0,
                       &thePromise,_fPartWrapper);
```

Fulfilling the Promise/Creating the Non-OpenDoc File

In your part editor's FulfillPromise, you should handle the case when the promise value needs to be fulfilled. The drag and drop object will put the file name requested by the target in the kODPropContents property under the kODFileType value.

```
ODValueType valueType = promiseSUView->GetType(ev);
if (ODISOStrEqual(valueType, "DRM_OS2FILE, DRF_UNKNOWN))
{
  CHAR szFile[CCHMAXPATH]
  ODStorageUnit* su = promiseSUView->GetStorageUnit(ev);
  su->Focus(ev,
            kODPropContents,
            kODPosUndefined,
            kODFileType,
            0,
            kODPosUndefined);
  long size = su->GetSize(ev);

  StorageUnitGetValue(su, ev, size, (ODValue)szFile);

  // Write the content to the file.
  ...

 SOM_ENDTRY
}
SOMFree(valueType);
```

Linking

This document contains a number of recipes for creating and maintaining links between content in OpenDoc parts. It assumes familiarity with

Data Interchange Basics, and references coding examples there. It also assumes you have read Clipboard Recipes; "Clipboard Recipes"; the basic clipboard recipes are augmented here for content kinds that support linking. Refer to the Class Documentation for a description of individual methods.

Note: Code appearing in these recipes has not been tested, and may contain errors.

Header Files

Parts that support linking need to include three header files:

```
#ifndef SOM_ODLinkSource_xh
#include <LinkSrc.xh>
#endif

#ifndef SOM_ODLink_xh
#include <Link.xh>
#endif

#ifndef SOM_ODLinkSpec_xh
#include <LinkSpec.xh>
#endif
```

Utility Functions Used in Code Samples

To hide the details of passing ODByteArray parameters to ODStorageUnit methods, the code samples use the following utility functions:

```
StorageUnitSetValue(ODStorageUnit* su,
                    Environment* ev,
                    ODULong size,
                    ODPtr buffer);
StorageUnitGetValue(ODStorageUnit* su,
                    Environment* ev,
                    ODULong size,
                    ODPtr buffer);
```

These functions wrap their arguments into an ODByteArray and make the call to ODStorageUnit::SetValue or ODStorageUnit::GetValue.

Associating Links with Content

Parts are responsible for maintaining the association between link and link source objects and the linked content. The association is inherently part-specific, as it depends on the content model of a part.

Persistent Representation of Links

Because the information necessary to associate a link with a region of content is part specific, there is no universal standard for representing links in part content. However, all link representations need to include supplementary information in addition to a reference to the link or link source object.

For a source of a link, the update ID associated with the linked content should also be saved. This update ID may be different from the update ID of the link source object if the link is updated manually.

For a destination of a link, the supplementary information includes the fields from the ODLinkInfo structure: the creation and modification time, the update ID of the last content incorporated from the link, the autoUpdate setting, and the kind of content accessed from the link. In addition, if a translation is applied to the data at the destination, both the original kind and the kind translated into must be identified; the kind field of the ODLinkInfo structure should identify the destination kind, including any translation.

The format of a content kind defines the standard representation for links. This allows linked content to be transferred to another part that understands the same content kind, while maintaining the same characteristics as the original.

Implementing InitPartFromStorage

If a part supports linked content, its InitPartFromStorage method needs to verify persistent references to link source and link storage units, and ensure link source objects always identify the correct source part.

Even though parts strongly reference link and link source storage units, these references can become invalid during data interchange to maintain the link behavior users expect. InitPartFromStorage should use the IsValidStorageUnitRef method of the ODStorageUnit object to validate each link and link source reference before attempting to internalize the reference. If the reference is invalid, the part should eliminate the source or destination link from its content model, without alerting the user.

When a part internalizes a valid link source reference, it should call ODLinkSource::SetSourcePart to ensure that the object identifies the part maintaining the source of the link. The part owning the source content of the link will change if that content is moved to another part.

Parts typically do not register for update notification in their InitPartFromStorage method, although they may. See Registering for Update Notification for recommendations on when to register.

AcquireLinkFromFocusedSU reads a link reference, validates the reference, and internalizes the link object. The storage unit parameter must be focused to a reference to a link storage unit. If this routine returns kODNULL, the caller should remove the reference from the part's content; this is part specific. If this routine returns a non-null result, the caller is responsible for releasing the object.

Note: Because this routine internalizes a link object, it must not be called during a clone transaction.

```
SOM_Scope ODLink*  SOMLINK MyPartAcquireLinkFromFocusedSU (MyPart *somSelf,
                                        Environment *ev, ODStorageUnit* su)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart"," AcquireLinkFromFocusedSU");

  ODLink* link = kODNULL;
  ODStorageUnitRef aSURef;

  SOM_TRY

  StorageUnitGetValue(su, ev, sizeof(ODStorageUnitRef), &aSURef);

  if (su->IsValidStorageUnitRef(ev, aSURef))
  {
    ODID linkID = su->GetIDFromStorageUnitRef(ev, aSURef);
    link = su->GetDraft(ev)->AcquireLink(ev, linkID, kODNULL);
  }

  SOM_CATCH_ALL
  SOM_ENDTRY
}
```

AcquireLinkSourceFromFocusedSU is very similar to the preceding routine. In addition, it ensures that the link source object references the right part. *Note: Because this routine internalizes a link object, it must not be called during a clone transaction.*

```
SOM_Scope ODLinkSource* SOMLINK MyPartAcquireLinkSourceFromFocusedSU(
                                MyPart *somSelf,
                                Environment *ev,
                                ODStorageUnit* su)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart"," AcquireLinkSourceFromFocusedSU");

  ODLinkSource* linkSource = kODNULL;
  ODStorageUnitRef aSURef;
```

```
  SOM_TRY

  StorageUnitGetValue(su, ev, sizeof(ODStorageUnitRef), &aSURef);

  if (su->IsValidStorageUnitRef(ev, aSURef))
  {
    ODID linkSourceID = su->GetIDFromStorageUnitRef(ev, aSURef);
    linkSource = su->GetDraft(ev)->AcquireLinkSource(ev, linkSourceID);

    // Ensure the link source object references this part as the source
    //   of the link.
    linkSource->SetSourcePart(ev, somSelf->GetStorageUnit(ev));
  }

  SOM_CATCH_ALL
  SOM_ENDTRY
}
```

Implementing Externalize

In your part's Externalize method, ensure that the part's storage unit (or one it references) contains a persistent reference to the link and link source objects your part uses. Even though your part holds a reference to the internalized link or link source object, container suite objects are subject to garbage collection when a draft is saved. If there are no persistent references to an object when the draft is saved, the underlying container suite object can be garbage collected even though the object has been internalized and has a non-zero reference count. This should never be a problem for production parts, but as you are implementing linking be sure to write the externalization code early to avoid this potential problem.

About Cloning Linked Content

If your part supports linked content, it will need to clone link and link source objects. Example code in this document uses the MyCloneStrongReference routines described in the Data Interchange Basics document. Note that these routines do not specify a particular destination storage unit to clone into. In fact, whenever your part clones a link or link source object, it must not specify a specific storage unit ID to clone into. Your part must clone link or link source objects by specifying kODNULLID as the destination storage unit.

Incorporating Linked Content from the Clipboard

Content on the clipboard may contain links. When that content is incorporated, any links should be incorporated, too. The representation of links is dependent on the content kind. However, all parts need to validate references to link and link source objects before using them, and must ensure that link source objects reference the part maintaining the source content.

This document shows two approaches for incorporating linked content. The first way is the easiest, so it is attractive when you are just getting linking working. The disadvantage of the first way is that it copies more data than necessary, potentially a lot more. Eventually, you will probably want to switch to the second recipe. It requires a little more work to implement, but can be significantly more efficient.

Incorporating Linked Content the Easy Way

The simplest way to incorporate linked content is to clone the content storage unit into your draft, extract the data format you want and incorporate it, and release the cloned storage unit. It leaves one or more abandoned storage units in your draft that will need to be garbage collected, but it is straightforward to implement.

The example described below is an elaboration on the MyReadFromContentSU method described in the Data Interchange Basics document.

```
SOM_Scope void  SOMLINK MyPartMyReadFromContentSU(
```

```
                        MyPart *somSelf,
                        Environment *ev,
                        ODStorageUnit* contentSU,
                        ODFrame* targetFrame,
                        ODCloneKind cloneKind,
                        ODLinkStatus embeddedFrameLinkStatus)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","MyReadFromContentSU");

  // Placeholders to be replaced by part-specific data
  ODByteArray data = CreateByteArrayStruct(kODNULL,0);
  ODStorageUnitRef aStrongRef;
  ODID strongClonedID = kODNULLID;
  ODID weakClonedID = kODNULLID;

  // Begin a clone transaction
  ODDraft* myDraft = somSelf->GetStorageUnit(ev)->GetDraft(ev);
  ODDraftKey draftKey = 0;

  ODVolatile(myDraft);
  ODVolatile(draftKey);

  SOM_TRY
    draftKey = ContentSU->GetDraft->BeginClone(ev, myDraft),
                                    targetFrame,
                                    cloneKind);

    TRY
      contentSU->Focus(ev, kODPropContents, kODPosUndefined,
                       kMyContentKind, 0, kODPosUndefined);

      // Clone the content storage unit into this part's draft
      tempID = clipDraft->Clone(ev,
                                key,
                                contentSU->GetID(ev),
                                kODNULLID,
                                kODNULLID);

    CATCH_ALL
      myDraft->AbortClone(ev, draftKey);
      RERAISE;
    ENDTRY
    myDraft->EndClone(ev, draftKey);

    // Acquire a reference to the temporary storage unit.
    ODStorageUnit* tempSU = myDraft->AcquireStorageUnit(ev, tempID);

    // Focus to the desired content kind in the temporary storage unit.
    tempSU->Focus(ev,
                  kODPropContents,
                  kODPosUndefined,
                  kMyContentKind,
                  0,
                  kODPosUndefined);

    // Read content from the temporary storage unit,
    // and incorporate it into this part.
    // This is specific to the data format of kMyContentKind,
    // which is not described here.
    ...

    // If a persistent link reference is encountered, read the reference,
    //   check its validity, and internalize the link object.

    ODLink* link = somSelf->AcquireLinkFromFocusedSU(ev, tempSU);
    if (link != kODNULL)
    (
      // Add link to this object's content model
      ...
  )
    else
    {
      // The link was broken during the clipboard transfer to maintain the link
      //   behavior users expect.  Do not incorporate this link into this part.
    }

    // If a persistent link source reference is encountered, read the reference,
    // check its validity, and internalize the link source object.
    // Its also important to ensure that the link source references
    // its new source part.

    ODLinkSource* linkSource = somSelf->
```

```
                                AcquireLinkSourceFromFocusedSU(ev, tempSU);
    if (linkSource != kODNULL)
    {
      // Add linkSource to this object's content model
      ...
  }
    else
    {
      // The link was broken during the clipboard transfer to maintain the link
      //   behavior users expect.  Do not incorporate this link into this part.
    }

    // Release temporary storage unit.
    // As there are no persistent references to this storage unit,
    // it can be garbage collected when the draft is closed.
    tempSU->Release(ev);

    // If the incorporated content contains embedded frames,
    // you must ensure certain frame characteristics are set properly.
    // Also, this part must remember the in-limbo status of each
    // embedded frame in its undo action data to implement
    // undo correctly.
    ODBoolean embeddedFrameWasInLimbo = embeddedFrame->IsInLimbo(ev);
    embeddedFrame->SetInLimbo(ev, kODFalse);
    embeddedFrame->SetContainingFrame(ev, targetFrame);
    embeddedFrame->ChangeLinkStatus(ev, embeddedFrameLinkStatus);

  SOM_CATCH_ALL
  SOM_ENDTRY
  DisposeByteArrayStruct(data);
}
```

Incorporating Linked Content the Efficient Way

A more efficient method of incorporating linked content from the clipboard is to read only the content kind you are pasting into your part. If the part that placed the content on the clipboard wrote promises for all content kinds it can provide, this method will result in the transfer of only one representation between the two parts.

The difficulty with this recipe is that your part has to hold the object IDs of the link and link source objects it clones until cloning is completed by calling EndClone. (You can convert a valid object ID into a reference during cloning, but if the ID is invalid, there's no way to represent that as a reference.) Only then can you determine if the IDs are valid and turn them into storage unit references or internalized objects.

The method show below uses the routine MyCloneStrongReference described in the Data Interchange Basics document.

```
SOM_Scope void  SOMLINK MyPartMyReadFromContentSU(
                        MyPart *somSelf,
                        Environment *ev,
                        ODStorageUnit* contentSU,
                        ODFrame* targetFrame,
                        ODCloneKind cloneKind,
                        ODLinkStatus embeddedFrameLinkStatus)
{
  MyPartData *somThis = MyPartGetData(somSelf);
  MyPartMethodDebug("MyPart","MyReadFromContentSU");

  // Placeholders to be replaced by part-specific data
  ODByteArray data = CreateByteArrayStruct(kODNULL,0);
  ODStorageUnitRef aLinkRef;
  ODStorageUnitRef aLinkSourceRef;
  ODID aLinkID = kODNULLID;
  ODID aLinkSourceID = kODNULLID;

  // Begin a clone transaction
  ODDraft* myDraft = somSelf->GetStorageUnit(ev)->GetDraft(ev);
  ODDraftKey draftKey = 0;

  ODVolatile(myDraft);
  ODVolatile(draftKey);

  SOM_TRY

    draftKey = ContentSU->
```

```
            GetDraft->
            BeginClone(ev, myDraft), targetFrame, cloneKind);

  TRY

    contentSU->Focus(ev, kODPropContents, kODPosUndefined,
                     kMyContentKind, 0, kODPosUndefined);

    // For demonstration, read the data into an ODByteArray.
    contentSU->GetValue(ev, contentSU->GetSize(ev), &data);

    // Link objects referenced from the value stream must be cloned to
    // the document draft.
    // This part must hold the link object ID until
    // EndClone is called.
    // aLinkSourceRef is a value from the byte array read above.
    aLinkID = somSelf->
              MyCloneStrongReference(ev, contentSU, aLinkRef, draftKey);

    // Link source objects referenced from the value stream
    // must also be cloned to the document draft.
    // aLinkSourceRef is a value from the byte array read above.
    aLinkSourceID = somSelf->
                    MyCloneStrongReference(ev,
                                           contentSU,
                                           aLinkSourceRef,
                                           draftKey);

  CATCH_ALL

    myDraft->AbortClone(ev, draftKey);
    RERAISE;

  ENDTRY

  myDraft->EndClone(ev, draftKey);

  // After EndClone has completed without error,
  // the IDs of cloned link and link source objects can be
  // converted to storage unit references or internalized.
  // The link or link source may have been broken during
  // the clipboard transfer to maintain the link behavior users
  // expect, so the cloned ID must be checked for validity.

  if (myDraft->IsValidID(ev, aLinkID))
  {
    ODLink* aLink = myDraft->AcquireLink(ev, aLinkID, kODNULL);
    // Add aLink to this object's content model
    ...
  }

  if (myDraft->IsValidID(ev, aLinkSourceID))
  {
    ODLinkSource* aLinkSource = myDraft->AcquireLinkSource(ev, aLinkSourceID);
    // Add aLinkSource to this object's content model
    ...
  }

  // If the incorporated content contains embedded frames,
  // you must ensure certain frame characteristics are set properly.
  // Also, this part must remember the in-limbo status of each
  // embedded frame in its undo action data to implement
  // undo correctly.
  ODBoolean embeddedFrameWasInLimbo = embeddedFrame->IsInLimbo(ev);
  embeddedFrame->SetInLimbo(ev, kODFalse);
  embeddedFrame->SetContainingFrame(ev, targetFrame);
  embeddedFrame->ChangeLinkStatus(ev, embeddedFrameLinkStatus);

SOM_CATCH_ALL

SOM_ENDTRY

DisposeByteArrayStruct(data);
}
```

Copying a Single Embedded Frame at the Source or Destination of

Suppose your part maintains a link source that consists of a single embedded frame. Or, suppose your part maintains a link destination where the content from the link is embedded. In either case, your part maintains the source or destination of the link, but the content comes from, or is, embedded as a separate part.

If the user selects such an embedded frame and copies it to the clipboard or drag and drop object, your part should follow Copying a Single Embedded Frame to the Clipboard. When your part adds the kODPropProxyContents property to the data transfer storage unit, the value (or values) you write should include a reference to a link or link source object you clone to the data transfer draft. In this way, if the embedded frame is copied or moved, the link source or destination will also be copied or moved if the part performing the paste or receiving the drop understands a value in the proxy content property.

When copying a single embedded frame at the source of a link, your part should also write a link spec, so additional destinations of the existing link source may be created. In this case, a part creating a new destination will not incorporate the proxy content on the clipboard or drag and drop object. When your CreateLink method is called to get the existing link, the proxy content available through the link will not contain the link source.

Posting a Link Specification to the Clipboard or Drag and Drop Obje

When data is copied to the clipboard or drag and drop object, the part should usually write a link specification in addition to content. The link specification indicates to any part performing a paste or drop that a persistent link may be created to the original content, using any of the representations present in the kODPropContents property.

The data in a link spec is private to the part writing the spec. The data written to the spec will be returned to the part via its CreateLink method if a link is actually created. The part needs to be able to identify the selected content from the link spec data. Because the link spec is only valid during the lifetime of the part, the data can contain pointers to information maintained by the part.

This example shows adding a link specification to the clipboard. To add a link spec to a drag and drop object, just replace clipContentSU with the content storage unit of the drag and drop object.

```
ODClipboard* clipboard = mySession->GetClipboard(ev);
ODStorageUnit* clipContentSU = clipboard->GetContentStorageUnit(ev);

ODLinkSpec* linkSpec = kODNULL;
ODVolatile(linkSpec);

TRY

  clipContentSU->AddProperty(ev, kODPropLinkSpec);

  linkSpec = myDraft->CreateLinkSpec(ev,
                                     somThis->fPartWrapper,
                                     link spec byte array);

  linkSpec->WriteLinkSpec(ev, clipContentSU);

CATCH_ALL

  ODReleaseObject(linkSpec);
  RERAISE;

ENDTRY

ODReleaseObject(linkSpec);
```

When Not To Post a Link Specification

When copying data to a content storage unit of the clipboard or when using the drag and drop operation, do not write a link specification if the data being copied is from a region that cannot be edited. Parts should not write a link specification in any of the following cases:

- The content storage unit belongs to a link source object, not the clipboard or drag and drop object.
- The part's draft does not have write permission. Before writing a link spec, check the draft permissions. If the permissions do not allow writing, a part should not write a link spec. Since the draft cannot be saved, the internal links cannot be preserved, and an external link created to another document will be abandoned when the draft is closed.

- The data is in a link destination maintained by the part.
- The part itself is embedded in a link destination (the link status of its display frame is kODInLinkDestination).
- The data contains only a portion of an existing link destination. The difficulty is maintaining the source link when the destination link is updated. It is in general impossible to determine what content the source link should assume when the destination changes.
- The data includes an entire link destination exactly. In this case, copying the content and pasting will create a another link destination if pasted into the same document (unless the user picks a lower-fidelity kind that does not support linking). If the part wrote a link specification and the new destination chose to create a link, an second link would be created between the existing and new destinations, probably not what the user wanted. The user should have to select the source to create a destination in another document.

Removing a Link Spec from the Clipboard

It is the responsibility of the part writing a link spec to remove it from the clipboard when:

- It becomes infeasible to create the link, for example, because the potential source content is deleted.
- The part's ReleaseAll() method is called.

In order to determine whether a link spec written by a part still resides on the clipboard, the part must remember the update ID of the clipboard when it writes the link specification. The following code fragment can be used to remove a link spec:

```
ODArbitrator* arbitrator = mySession->GetArbitrator(ev);
ODFrame* clipboardOwner = arbitrator->AcquireFocusOwner(ev, clipboardFocus);
if ((activeFrame == clipboardOwner) ||
 (arbitrator->RequestFocus(ev, clipboardFocus, activeFrame)))
{
  ODClipboard* clipboard = mySession->GetClipboard(ev);

  TRY
    if (myClipboardUpdateID == clipboard->GetUpdateID(ev))
    {
      ODStorageUnit* clipContentSU = clipboard->GetContentStorageUnit(ev);

      // Focus to the link spec property
      clipContentSU->Focus(ev,
          kODPropLinkSpec,
          kODPosUndefined,
          kODNULL,
          0,
          kODPosUndefined);

      // Remove the property
      clipContentSU->Remove(ev);
    }

  CATCH_ALL

    // Ignore errors.

  ENDTRY
}
ODReleaseObject(ev, clipboardOwner);
```

Specifying Kinds Supported via nmap Resources

The settings and choices in the Paste As dialog rely on accurate information about the kinds supported by a part. See the Binding recipe for details on how to specify the kinds supported by a part.

Implementing Paste with Link (Incorporating Content)

When the user chooses the **Paste As** menu item, the part should call the clipboard's ShowPasteAsDialog method to display the dialog. If the ODPasteAsResult value set by that method specifies a link should be created to the source, this recipe should be followed.

Parts should enable the **Paste As** menu item only if the draft's permissions allow writing.

Basically, the receiving part should ignore the content on the clipboard, and read the link spec value there using ODLinkSpec::ReadLinkSpec. The link object is obtained via the receiving part's draft's AcquireLink method.

The routine MyPasteWithLink shown here takes three parameters. The first is the content storage unit of the clipboard or drag and drop object. The second is the structure returned by the ShowPasteAsDialog method of either the clipboard or drag and drop object. The third is a boolean indicating whether content is pasted from the clipboard or the drag and drop object; this makes a difference in how the routine adds undo actions.

MyPasteWithLink shows how the part performing the paste is expected to add undo history. Because link creation involves both the source and destination parts, an undo transaction is used to capture actions by both parts. If the link is being created from the clipboard, the destination must begin an undo transaction before calling AcquireLink. If the link is created during a drop, the part that initiated the drop is assumed to have already started an undo action, so this part simply adds its undo data as a single action.

Note that MyPasteWithLink will register for update notification even if the user has requested manual updates. Because cross-document links may be created asynchronously, its not always possible to read data from a link immediately. Registering for notification gives the source part a chance to deliver the data. When this part's LinkUpdated method is called, the part should unregister if the destination updates manually.

For simplicity, translation of content incorporated through a link is not covered here. Examples of translation appear in the section "Embedding a Part via the Paste As Dialog" in Data Interchange Basics.

```
void MyPasteWithLink(Environment *ev,
  ODStorageUnit* contentSU,
  ODPasteAsResult pasteAsResult,
  ODBoolean fromClipboard)
{
  ODDraft* myDraft = fSOMSelf->GetStorageUnit(ev)->GetDraft(ev);
  ODLinkSpec* linkSpec = kODNULL;

  ODIText* undoActionName = kODNULL;
  ODIText* redoActionName = kODNULL;
  ODActionData noActionData = (ODActionData) CreateByteArrayStruct(kODNULL,0);

  ODUndo* undo = fSOMSelf->GetStorageUnit(ev)->GetSession(ev)->GetUndo(ev);

  // For demonstration, locals are used; your part
  // must integrate these into its content
  ODLinkInfo linkInfo;
  ODLink* link = kODNULL;

  SOM_TRY

    // Initialize text for undo.
    // Not localizable!
    undoActionName = CreateIText(smRoman, langEnglish, "Undo Paste");
    redoActionName = CreateIText(smRoman, langEnglish, "Redo Paste");

    // Read the link spec from the content storage unit.
    // The link spec is created with null ODPart and ODByteArray
    // pointers since it is initialized by ReadLinkSpec.

    linkSpec = myDraft->CreateLinkSpec(ev, kODNULL, kODNULL);

    // Focus to the link spec property; ReadLinkSpec will focus
    // to the appropriate value.
    contentSU->Focus(ev,
         kODPropLinkSpec,
         kODPosUndefined,
         kODNULL,
         0,
         kODPosUndefined);

    linkSpec->ReadLinkSpec(ev, contentSU);

    // Establish the link
    TRY

      // If this is a paste from the clipboard, start an undo transaction
      if (fromClipboard)
      {
```

```
        undo->AddActionToHistory(ev,
                                 somThis->fPartWrapper,
                                 &noActionData,
                                 kODBeginAction,
                                 undoActionName,
                                 redoActionName);
      }

      link = myDraft->AcquireLink(ev, kODNULLID, linkSpec);

      linkInfo.change = kODUnknownUpdate; // not yet updated
      GetDateTime(&linkInfo.creationTime);
      linkInfo.kind = ODISOStrFromCStr(pasteAsResult.selectedKind);
      linkInfo.autoUpdate = pasteAsResult.autoUpdateSetting;

      // Add link and linkInfo to the content model of this part.
      //   This is part specific.
      ...

      // Add this action to the undo stack.
      // If this is a paste from the clipboard, end the undo
      // transaction started above.
      // Otherwise, this must be a drop, so add to the existing transaction.
      ODActionData* actionData = ...
      undo->AddActionToHistory(ev,
                               somThis->fPartWrapper,
                               actionData,
                               (fromClipboard ? kODEndAction : kODSingleAction),
                               undoActionName,
                               redoActionName);

    CATCH_ALL

      // Abort the undo transaction.
      if (fromClipboard)
      {
        undo->AbortCurrentTransaction(ev);
      }
      RERAISE;

    ENDTRY

    // Always register for notification that the initial link content
    // is available.
    // If the user specified manual updating, this part should
    // unregister after LinkUpdated is called.
    if (!fSOMSelf->IsRegistered(ev, link))
    {
      // RegisterDependent will call LinkUpdated.
      link->RegisterDependent(ev, somThis->fPartWrapper, linkInfo.change);
    }
    else
    {
      // Get update now
      fSOMSelf->LinkUpdated(ev, link, link->GetUpdateID(ev));
    }

  SOM_CATCH_ALL

  SOM_ENDTRY

  DisposeIText(undoActionName);
  DisposeIText(redoActionName);
  delete linkSpec;
}
```

Implementing Paste with Link (Embedding Content)

Any part that supports embedding and linking should allow the user to embed content from the clipboard and create a link. When content from a link is embedded, the destination part (the part performing the paste) does not clone the content storage unit of the clipboard or drag and drop object. Instead, the part reads the link spec from the content storage unit, acquires the link from the link spec, and creates an embedded part by cloning the content storage unit of the link. When the link is updated, the destination part discards the embedded part and creates a new embedded part from the new link content. The embedded part is not involved in the maintenance of the link. The link border appears around the embedded part's frame and is drawn by the destination part.

When content is embedded, the user may choose a particular content kind or specify a translation to another kind. "Embedding a Part via the Paste As Dialog" in Data Interchange Basics describes the steps the part performing the paste takes to embed the desired content kind. The recipe is directly applicable to embedding linked content; in this case, the content storage unit of the link is cloned (as above), and then any necessary translation is performed.

Drop Result when Pasting a Link

When a link is created using drag and drop, be sure your part's Drop method returns the ODDropResult value kODDropCopy.

Registering for Update Notification

The "Implementing Paste with Link" recipes demonstrate how a destination part can register with the link to receive automatic notifications when the link is updated with new content. When a new link destination is created, parts should always register to receive notification that the initial content is available.

Parts register for updates by calling the RegisterDependent method of the link method. The part supplies the update ID of the content it last read from the link. To get the initial content from the link, a part can use the update ID constant kODUnknownUpdate to ensure it receives notification. Parts receive notification of an update via their LinkUpdated method.

If a part's LinkUpdated method is called after registering to receive the initial content from a link, the part should unregister if the user specified that the destination should update manually.

Parts should register for notification when a part is internalized if the link updates automatically and the link is visible or may affect layout of the part. When your part calls RegisterDependent, its LinkUpdated method may be called before RegisterDependent returns, so be sure to call RegisterDependent only when your part is prepared to receive a notification. Parts may unregister and re-register for notification as they wish.

If a part is in a read-only draft, you may call RegisterDependent, but your part will not receive notification (your part's LinkUpdated method will not be called).

Undoable Actions and Linking

The following user actions involving links should be undoable (this list may not be exhaustive):

- Pasting content creating a link to the source.
- Pasting content containing existing links.
- Breaking a link at the source or destination via the Link Info dialog.
- Deleting a selected link and its content, or deleting content containing one or more source or destination links.
- Cutting content containing one or more source or destination links.

The updating of links, at the source or destination, is not an undoable user operation. A part updating the destination of a link does not have to create an undo action to restore its previous content. Similarly, a part updating a link from source content does not need to create an action state to restore the link to its previous contents.

When content containing a source link is removed by deleting or cutting, or when a source link is broken, your part should:

- Save its reference to the link source object in undo action data and disassociate the link source from the content in whatever way is appropriate for the data format,
- Call linkSource->SetSourcePart(ev, kODNULL); so the object no longer references the part.

Note that your part should generally not update a removed or broken link source with empty content; just leave the link source in its existing state.

If the action is undone, the part should:

- Reinstate the link source object into your parts content,
- Call `linkSource->SetSourcePart(ev, somThis->GetStorageUnit(ev));` to reestablish this part as the source of the link.

If a link or link source is broken, your part must change the link status of any affected frames; see the Frame Link Status recipe.

Similarly, links may be broken at their destination, or they may be deleted or cut along with the content at the destination of the link. When this happens, your part should:

- Save a reference to the link object in undo action data, and, if the link was explicitly broken, disassociate the link from the content.
- If this part is registered to receive update notifications, call `link->UnregisterDependent(ev, somSelf->fPartWrapper)` if this was the last automatically updated destination of the link in this part.

Your part should hold a reference to each link or link source objects in undo action data until the part's DisposeActionState method is called.

Undoing the Paste or Drop of Linked Content

When designing and implementing undo for pasted or dropped content, you must be take into account the behavior of destination links in the pasted or dropped content. Your part does not treat updating of a link destination as an undoable action. When a link updates, your part removes the existing content at the destination, and replaces it with new content from the link. Your part does not create an undo action, and does not save the content that is replaced. If the modification at the source of the link is undone, your part will receive another link update, putting the destination into an equivalent state as before but with different objects. Now suppose the user undoes the paste or drop. Your part must not assume the same objects that were created at the time of the paste are present. Your part must undo the paste with whatever objects are currently in place. In particular, the undo information your part associates with the paste or drop should not include specific objects cloned from links.

Accessing a Link

Because source and destination parts may attempt to access a link simultaneously, parts must acquire a lock in order to get the storage unit containing the content of the link. Many methods of classes ODLink and ODLinkSource require a valid link key that can only be obtained by successfully acquiring a lock first. Parts must not access the storage unit returned by GetContentStorageUnit after unlocking the link.

Basic Recipe for Writing to a Link

This basic recipe can be used to write the initial content into a link when created, and to update the link when the source content changes. This example routine illustrates the use of promises, which must be fulfilled by this part's FulfillPromise method. It uses the routine MyWriteToContentSU described in the Data Interchange Basics document.

To ensure that promises written into the link are fulfilled correctly, this routine adds a kODPropCloneKindUsed property before calling MyWriteToContentSU. In their FulfillPromise methods, all parts should check for this property to see if a clone kind other than kODCloneCopy should be used. When FulfillPromise is called to write to a link, BeginClone will fail with an inconsistent clone kind error if the clone kind is other than kODCloneToLink.

When the link is initially created or when its being updated, the justRewriting parameter should be kODFalse. If the part's CreateLink method is called to create another destination of an existing link, and the content at the source of the link has not changed since the link was last updated, the justRewriting parameter should be kODTrue. See Implementing CreateLink for more information on when to pass kODTrue to the justRewriting parameter.

The promiseData parameter is whatever information the part needs to identify the promised content when its FulfillPromise method is called.

The contentShape parameter will be written into the link in the suggested frame shape annotation. This property will be used as the frame shape should content be embedded at the destination of the link.

The partName parameter is passed thru to MyWriteToContentSU.

The Clear method removes the kODPropContents property from the link's content storage unit. After calling Clear, your part must call GetContentStorageUnit, add the contents property, write promise values (or actual values) into the link, and call ContentUpdated to notify destinations. After your part calls Unlock, OpenDoc will fulfill any promises for values that are used by destinations.

Note that the Clear and ContentUpdated methods of ODLinkSource will return an exception if passed kODUnknownUpdate as the argument to the update parameter.

```
void MyWriteToLink (Environment *ev,
```

```
  ODLinkSource* linkSource,
  ODUpdateID updateID,
  ODBoolean justRewriting,
  ODByteArray* promiseData,
  ODShape* contentShape,
  ODIText* partName)
{
  ODVolatile(linkSource);

  ODLinkKey linkKey;
  ODVolatile(linkKey);

  if (linkSource->Lock(ev, 0, &linkKey))
  {
    // Remember the previous updateID in case updating fails
    ODUpdateID previousID = linkSource->GetUpdateID(ev);

    TRY
      // Call Clear to remove the contents property and allow writing promises.
      // If justRewriting is true, use the existing updateID of the link
      linkSource->Clear(ev, (justRewriting ? previousID : updateID), linkKey);

      ODStorageUnit* linkContentSU = linkSource->GetContentStorageUnit(ev, linkKey);

      // Add the clone kind used property so promises are
      // fulfilled using kODCloneToLink
      ODSetULongProp(ev,
                     linkContentSU,
                     kODPropCloneKindUsed,
                     kODCloneKind,
                     kODCloneToLink);

      fSOMSelf->MyWriteToContentSU(ev,
                                   linkContentSU,
                                   kODCloneToLink,
                                   promiseData,
                                   contentShape,
                                   partName);

      // If the link is being updated, and not just rewritten, inform the link source
      //   that it has changed, so destinations can be notified.
      if (!justRewriting)
        linkSource->ContentUpdated(ev, updateID, linkKey);

    CATCH_ALL

      // Clear a partially-updated link.  Destinations must deal
      //   gracefully with a link content storage unit that has no
      //   contents property.
      linkSource->Clear(ev, previousID, linkKey);
      RERAISE;

    ENDTRY

    linkSource->Unlock(ev, linkKey);
  }
}
```

Implementing CreateLink

The creation of a link is initiated when the part performing a paste or drop calls its draft's GetLink method, passing in a link spec read from the clipboard or drag and drop container. A recipe for pasting a link appears elsewhere in this document.

A link source is created by a part's CreateLink method. Any content kind written to the clipboard or drag and drop container should be available through the link. When providing the initial content for a link, a part may write either promises or actual data. By writing promises, a part can provide content for only the data kind(s) actually used by destinations of the link. When a destination reads data through the link, the source part will be called to fulfill its promise for a particular content kind. A separate document discusses promises in more detail.

This example implementation of CreateLink demonstrates how your part should add to the undo history when a new link source is created. The action will be added to an undo transaction (started by the part performing the paste, or this part if the operation is a drop). See Better Undo Recipe for CreateLink for the preferred way to handle undoing the link creation, although it is a little more work for your part.

CreateLink should only be called by drafts, not directly by parts. This example demonstrates how your part may implement its CreateLink method. It uses the routine MyWriteToLink to create the initial content of the link, using promises, and to ensure that all values are present each time an additional destination is created. If your part does not write promises to a link, the else-clause in this example can be eliminated.

```
SOM_Scope ODLinkSource* SOMLINK MyPartCreateLink(
                                MyPart *somSelf,
                                Environment *ev,
                                ODByteArray* data)
{
  MyPartData *somThis = MyPartGetData(somSelf);

  ODLinkSource* linkSource = kODNULL; ODVolatile(linkSource);

  // Placeholders to be replaced by part-specific data
  ODByteArray* promiseData;
  ODShape* contentShape;
  ODIText* partName;    // Optional
  ODUpdateID sourceUpdateID;

  SOM_TRY

    // Call a part-specific method to get the link source if it already exists
    linkSource = somSelf->MyGetExistingLinkSource(ev, data);

    if (linkSource == kODNULL)
    {
      ODSession* session = somSelf->GetStorageUnit(ev)->GetSession(ev);

      // Call this part's draft to create the link source object
      ODDraft* myDraft = somSelf->GetStorageUnit(ev)->GetDraft(ev);
      linkSource = myDraft->CreateLinkSource(ev, somThis->fPartWrapper);

      // Add the link to the content model of this part.
      //   This is part specific.
      ...

      // Associate a new update ID with the source content.
      sourceUpdateID = session->UniqueUpdateID(ev);

      TRY

        MyWriteToLink (ev,
            linkSource,
            sourceUpdateID,
            kODFalse,
            promiseData,
            contentShape,
            partName);

      CATCH_ALL

        // Remove the link from the content model of this part
        //...

        linkSource->Release(ev);
        linkSource = kODNULL;
        RERAISE;

      ENDTRY;

      // A new link source was successfully created,
      // so add an action to the undo stack.
      // The action names should never appear as menu items if
      // the parts implement undo properly.
      // The action data is part specific.
      ODIText* undoActionName = CreateIText(smRoman,
                                            langEnglish,
                                            "Undo Create Link");
      ODIText* redoActionName = CreateIText(smRoman,
                                            langEnglish,
                                            "Redo Create Link");
      ODActionData* actionData = ...
      session->GetUndo(ev)->AddActionToHistory(ev,
                                               somThis->fPartWrapper,
                                               actionData,
                                               kODSingleAction,
                                               undoActionName,
                                               redoActionName);
      DisposeIText(undoActionName);
      DisposeIText(redoActionName);
    }
```

```
    else
    {
      // If your part initialized the link with promises, add promises
      // for value kinds not already present in the link because
      // current destinations did not use them.

      // If the source content has not changed since the link was last updated,
      // rewrite the link, otherwise, update it now.
      // The variable sourceUpdateID contains the update ID currently
      // associated with the content at the source of the link.
      ODBoolean justRewriting = (sourceUpdateID == linkSource->GetUpdateID(ev));

      TRY

        MyWriteToLink (ev,
            linkSource,
            sourceUpdateID,
            justRewriting,
            promiseData,
            contentShape,
            partName);

      CATCH_ALL

        // If the link could not be updated, CreateLink should fail rather than
        //   return a linkSource without a contents property.
        linkSource = kODNULL;
        RERAISE;

      ENDTRY;
    }

    // The result of this method must be released by the caller.
    linkSource->Acquire(ev);

  SOM_CATCH_ALL

  SOM_ENDTRY

  return linkSource;
}
```

Better Undo Recipe for CreateLink

If your part implements undo support as described in the recipe above, and the link is created via cross-document drag and drop, the undo action stack will contain two separate undo items; one for the drop, and another for the creation of the link source. There are two undo items in this case because the call to the source part's CreateLink method is postponed until after the drag completes and returns to the source part.

Your part can avoid introducing a second undo item in this case by doing the following. When you write a link spec to a drag and drop object's content storage unit, include in the part data information identifying the undo history transaction your part created for the drag and drop operation. Then, in your CreateLink method, check for this information, and instead of adding an undo action as described above, modify some private data your part associates with the drag and drop undo transaction so that the link creation is undone (or redone) as well.

When CreateLink Is Called To Get an Existing Link

CreateLink may be called multiple times with the same link spec to create multiple destinations of the same link. If your part writes actual content into the link, it can just return the link object. However, if your part writes promises, it must ensure all values are available in the link.

If CreateLink is called to get an existing link, its important that the link contain all value types available via the clipboard or drag and drop object. If a part writes promises into a link, the link will contain only actual content used by current destinations; if the draft is saved, unfulfilled promises will be removed. When CreateLink is called, the part must ensure that at least a promise is present for all value types. A part can handle this as shown in MyPartCreateLink above.

Parts can now use the Clear method to replace promises in a link without notifying existing destinations. Clear may be called with the existing update ID; by not calling ContentUpdated, destinations will not be notified (as shown in MyPartCreateLink). If ContentUpdated is called with the existing update ID, the circular link update alert will appear.

Updating a Link

Your part can use the MyWriteToLink method to update an existing link. The updateID argument should be the updateID responsible for changing the content at the source of the link. The justRewriting parameter should be kODFalse. This recipe can be used to update both an automatic and a manual link.

In general, it is not necessary, and sometimes undesirable, to update links immediately in response to a content change. When content is changing due to keyboard events, for example, it is reasonable to wait for a pause before updating any affected links.

Creating or Updating a Link Consisting of a Single Embedded Fram

If your part creates or updates a link source consisting of a single embedded frame, your part calls the embedded part's CloneInto method to supply the link content. To allow the embedded part to promise its content, the embedded part must know to use the ODCloneKind value kODCloneToLink (instead of kODCloneCopy) in its call to BeginClone in its FulfillPromise method. Before calling the embedded part's CloneInto method, your part should add a kODPropCloneKindUsed property to the link source object's content storage unit, add the value type kODCloneKind, and write the value kODCloneToLink as a 32-bit integer. For example:

```
ODSetULongProp(ev,
               contentSU,
               kODPropCloneKindUsed,
               kODCloneKind,
               kODCloneToLink);
```

This is demonstrated in the example routine MyWriteToLink in this document.

Notes on Implementing FulfillPromise

All parts should check for the presence of a kODPropCloneKindUsed property in their FulfillPromise method. The following code can be used to determine the correct clone kind argument to BeginClone:

```
ODCloneKind cloneKind = kODCloneCopy;
ODStorageUnit* promiseSU = promiseSUView->GetStorageUnit(ev);
if (ODSUExistsThenFocus(ev,
                        promiseSU,
                        kODPropCloneKindUsed,
                        kODCloneKind))
   cloneKind = ODGetULongProp(ev,
                              promiseSU,
                              kODPropCloneKindUsed,
                              kODCloneKind);
```

The default clone kind is kODCloneCopy unless a specific clone kind is specified by a kODPropCloneKindUsed property.

Updating the Destination of a Link

A part can update the destination of a link in its LinkUpdated method, which is called automatically if the part registers as a dependent of a link, or in response to the user's request to update a manual link. The example method MyUpdateLinkDestination provides the skeleton structure; add code to incorporate or embed the data into your content. You should be able to use a common routine to integrate data whether it comes from a link, the clipboard, or a drop.

If the last update at the source of a link failed, leaving a link without content, destinations that try to read the link will get the error kODErrNoLinkContent back from GetContentStorageUnit. This ensures that a destination cannot attempt to embed a storage unit without

content as a part. Your part should consider this a temporary problem and refrain from updating at this time. The MyUpdateLinkDestination routine shown here will raise this exception to the caller, which should just ignore the error.

If, however, your part gets back the error kODErrCannotEstablishLink from GetContentStorageUnit, this means that the link could not be established correctly. This error will only be returned after a link has been returned by your draft's AcquireLink method, but before any content is available through it. In response to this error, your part should remove the link from its content model, release its reference to the link, and alert the user.

Your part may encounter invalid persistent references to link or link source objects when updating from a link. The appropriate response is to ignore them and treat the content they were associated with as unlinked content.

Note that updating the destination of a link is not an undoable operation. You should not create an undo action for it.

For future compatibility, this recipe does not assume the link content storage unit is in the same draft as the destination part.

```
void MyUpdateLinkDestination(Environment *ev,
                             ODLink* link,
                             ODLinkInfo* linkInfo)
{
  AppleTestDrag_DragText *fSOMSelf;
  AppleTestDrag_DragTextData *fSOMThis =
       AppleTestDrag_DragTextGetData(fSOMSelf);

  ODLinkKey linkKey;

  ODVolatile(link);
  ODVolatile(linkKey);

  ODFrame* myDisplayFrame;
  ODFrame* anEmbeddedFrame;

  if (link->Lock(ev, 0, &linkKey))
  {
    TRY

      ODStorageUnit* linkContentSU = link->GetContentStorageUnit(ev, linkKey);
      ODDraft* sourceDraft = linkContentSU->GetDraft(ev);

      // Remove the current content at the destination of the link.
      //   This is part specific.
      ...

      // Insert content from the link into the destination.
      fSOMSelf->MyReadFromContentSU(ev,
          linkContentSU,
          kODNULL,
          kODCloneFromLink,
          kODInLinkDestination);

      // Update the link info for this destination
      linkInfo->change = link->GetUpdateID(ev);
      linkInfo->changeTime = link->GetChangeTime(ev);

    CATCH_ALL

      link->Unlock(ev, linkKey);
      RERAISE;

    ENDTRY

    link->Unlock(ev, linkKey);

    // Propagate changes to source links maintained by this part.
    //   This, too, is part specific.
    ...

    // Any time this part's content is changed, notify all containing parts.
    //   This method should be called for all frames displaying the changed
    //   content.
    myDisplayFrame->ContentUpdated(ev, linkInfo->change);

    // Make sure that this change is saved when the document is closed
    fSOMSelf->GetStorageUnit(ev)->GetDraft(ev)->SetChangedFromPrev(ev);
  }
}
```

Link Source Moved Across Parts

OpenDoc allows content to be moved across parts; if the content contains links, the links are preserved if the destination uses a format supporting links. (If the destination uses a format that does not specify linking, links will be broken at the source.) If content is moved to a part that supports different content kinds, it may not be able to write all kinds into the link that the original source part could. Parts do not need to record content kinds they write to a link as part of the information they associate with a link source. When its time to update the link, they should just write the kinds irrespective of what might already be in the link. The Clear method of ODLinkSource will ensure that any obsolete value kinds are removed from the link.

At the destination of a link, a part must be prepared to deal with the content kind missing from the link. If possible, the part should attempt to use another format from the link. Otherwise, the destination should break the link and leave the existing content in place.

Note: This function is not supported in the current release of OpenDoc.

Update IDs

All linked content, that is, every region of content at the source or destination of a link, has an update ID associated with it. The source part determines the update ID associated with the link. The destination part is responsible for remembering the update ID of the link at the time the destination last updated from the link.

Parts are responsible for correctly propagating update IDs to enable OpenDoc to detect circular links. If a circular link goes undetected, the parts involved can update each other indefinitely. When updating a link, the source part should reuse the update ID associated with the content causing the link to update. If the content change was due to an updated link destination, the update ID of the link destination should be adopted as the update ID of the affected link.

If a content change originates in a part, such as in response to keyboard events, all links directly affected should be associated with the same new update ID. A new update ID is created by calling the UniqueUpdateID method:

```
ODUpdateID updateID =
  somSelf->GetStorageUnit(ev)->GetSession(ev)->UniqueUpdateID(ev);
```

Your part must not create a distinct update ID for each affected link source; use the same ID for all of them.

Links Containing Embedded Frames

Links may be established to content containing embedded frames, just as content containing embedded frames may be copied and pasted without a persistent link. The destination of such a link contains a copy of the parts embedded at the source of the link. When the destination is updated, the previously embedded parts are discarded, and replaced by new ones cloned from the link. Display frames for the new parts are hooked into the frame hierarchy at the destination, and facets are created for the currently visible frames.

One special case is a link to one embedded frame. If the user copies a single embedded frame to the clipboard, the active part should clone the embedded part into the content storage unit of the clipboard, and then create a kODPropProxyContent property in the same content storage unit in which container-specific annotations about the embedded frame are written (see the Data Interchange Basics recipes for details). If a paste with link is performed at the destination, a link is created to the embedded frame, without involving other intrinsic content of the containing part. Should that embedded frame be again cut or copied to the clipboard, at either the source or destination of the link, the embedded part should be cloned into the content storage unit of the clipboard, as before. After the embedded part has cloned itself, the containing part should add a proxy content property to the same content storage unit, and write out a value that includes the link or link source (cloned to the clipboard and referenced by the proxy content). When the clipboard content is pasted into any part that understands the value type in the proxy content, the link can be preserved.

Frame Link Status

All frames have a link status that describes the frame's participation in links. The link status of a frame should be set to indicate whether it is

embedded in the source of a link, in the destination of a link, or not involved in any link. Parts can use this information to determine when they should not allow creation of a new link, or not allow content to be cut.

All parts must set the link status of frames they embed, even parts that do not otherwise support linking. Whenever frames are embedded during data interchange, the active part must set the link status of each embedded frame using the ChangeLinkStatus method, even if a link is not created. Also, a part's LinkStatusChanged method is responsible for notifying its embedded frames, as described below under "Implementing LinkStatusChanged."

When a link is created, a part should call ODFrame::ChangeLinkStatus for all of its frames that display data at the source or destination of the link. ChangeLinkStatus will change the value of the link status (ODLinkStatus type) after examining the link status of its containing frame. ChangeLinkStatus will call ODPart::LinkStatusChanged for its displayed part in order to give the part the chance to call ChangeLinkStatus for any of its embedded frames. Frames and parts can examine the link status of a frame by calling ODFrame::GetLinkStatus.

If incorporation adds new embedded frames to the receiving part, the link status of those embedded frames must be set by calling their ChangeLinkStatus method. If an embedded frame lies within the destination of a link maintained by this part, its status should be changed to kODInLinkDestination (as would be the case when this is a paste with link operation). Otherwise, if the embedded frame lies within the source of a link, its status should be changed to kODInLinkSource. (When an embedded frame is contained in both a link source and a link destination, its status should be set to kODInLinkDestination.) If the embedded frame is not within any link maintained by this part, its status should be changed to kODNotInLink.

When a part breaks a link, the part needs to call the ChangeLinkStatus method of all affected embedded frames. The new status can be just the embedded frame's status with respect to the part, since ChangeLinkStatus takes the containing frame's status into account.

Parts are responsible for ensuring that the link status of an embedded frame is correct when it internalizes the frame. Parts that do not support linking can simply call the ChangeLinkStatus method of the embedded frame, specifying kODNotInLink. Parts that do support linking should set the link status according to the embedded frame's participation in links that the part maintains. The frame object will do nothing if the status is unchanged, and will take the containing frame's status into consideration to simplify the part's responsibilities.

Implementing LinkStatusChanged

The LinkStatusChanged method is called to notify a part that one of its display frames was included in the creation or deletion of a link. The part must pass this notification along to any embedded frames by calling ODFrame::ChangeLinkStatus. As an optimization, embedded frames which are already involved in links managed by this part do not need to be notified, since their status does not change (if the embedded frame is in a destination maintained by your part, its still in a destination; if its in a source, a destination could not be created around it).

Content Change Protocol

Because linked content may contain embedded frames, some protocol between parts is required to inform containing parts of changes to embedded content. Two methods are involved when content changes:

```
void ODFrame::ContentUpdated(Environment* ev,
                      ODUpdateID change);

void ODPart::EmbeddedFrameUpdated(Environment* ev,
                      ODFrame* frame,
                      ODUpdateID change);
```

If a content change in a part affects one of its display frames, the part should call ODFrame::ContentUpdated, passing it the unique update ID for the change. This unique ID may be obtained through a call to ODSession::UniqueUpdateID if the change originated in the part. ODFrame::ContentUpdated will call ODPart::EmbeddedFrameUpdated for all containing parts in the frame hierarchy. The containing parts then know that some of its embedded content has changed. If the embedded part is involved in a link source, the containing part maintaining the link can choose to update the link with the new data.

Implementing EmbeddedFrameUpdated

The EmbeddedFrameUpdated method is called to notify a part that some change to the content of an embedded frame has occurred. If the part maintains the source of a link that includes the embedded frame, the part should update the link. The part is passed a reference to the embedded frame, and the update ID associated with the modification.

Calling ODFrame::ContentUpdated

When content displayed by a frame is changed, call the affected frame's ContentUpdated method. The display frame will pass the notification on to all containing parts, unless the frame is the root frame of the document.

Displaying Link Info Dialogs

Recipes for displaying the link info dialogs are included in the Info recipe document.

Displaying Link Borders

A border should be drawn around a link when the link is selected or when the "Show Links" setting is checked in the Document Properties notebook.

Whenever a part draws its content, it should call the ShouldShowLinks method of the facet's window. If ShouldShowLinks returns kODTrue, borders should be drawn.

```
if (myFacet->GetWindow(ev)->ShouldShowLinks(ev))
{
  // Draw borders around links in this facet.
}
```

Implementing RevealLink

Parts that create links need to implement the RevealLink method to enable users to navigate from the destination of a link to the source content. RevealLink may be called when the part's document is not the front-most process; bring the process to the front, if necessary. If the link cannot be made visible in an existing display frame, the part should open a new window.

This is the rough outline of a typical RevealLink implementation. It uses four methods not defined here: MySetFrontProcess, MyBestDisplayFrame, MyActivateFrame, and MyScrollToLink.

```
SOM_Scope ODLinkSource* SOMLINK MyPartRevealLink(MyPart *somSelf,
                                                 Environment *ev,
                                                 ODLinkSource* linkSource)
{
  MyPartData *somThis = MyPartGetData(somSelf);

  // Make sure this process is front-most
  somSelf->MySetFrontProcess(ev);

  // Choose a display frame for the source content.
  //   This is part specific.  If no suitable display frame exists,
  //   the part should open a new window

  ODFrame* frame = (ODFrame*) somSelf->MyBestDisplayFrame(ev, linkSource);

  // If the best frame has a containing frame, ensure the display
  //   frame is revealed.

  ODFrame* containingFrame = frame->AcquireContainingFrame(ev);
  if (containingFrame != kODNULL)
  {
    ODPart* containingPart = kODNULL;
```

```
    TRY
      containingPart = containingFrame->AcquirePart(ev);
      containingPart->RevealFrame(ev, frame, kODNULL);
    CATCH_ALL
    ENDTRY

    ODReleaseObject(ev, containingPart);
    ODReleaseObject(ev, containingFrame);
  }

  // Activate my frame by using my normal activation method.
  //   RevealLink may be called when the part's document is not the
  //   front-most process, so be sure to bring it to the front if
  //   necessary.

  somSelf->MyActivateFrame(ev, frame);

  // Scroll my frame as necessary to make the source content visible.

  somSelf->MyScrollToLink(ev, linkSource);
}
```

Editing in a Link Destination

If your display frame's link status is kODInLinkDestination (determined by calling ODFrame::GetLinkStatus), the frame is embedded in a link destination and editing is not usually allowed. If the user attempts to edit content in the frame, the part should call the EditInLink method of the display frame. In response, OpenDoc will call the EditInLinkAttempted method of the part maintaining the destination of the link. Parts that support linking and embedding need to override this method.

EditInLinkAttempted should check that it maintains a link destination including the argument embedded frame. If not, it should return kODFalse. If it does maintain a link destination, it should present an alert informing the user of the attempted edit to a link destination, and allow the user to find the source of the link or break the destination link In either case, the part should not activate one of its display frames. If the user chooses to break the link, the part should change the link status of all affected embedded frames. An alert for this purpose is included in the Example Part Resources document in the Sample Code folder.

ODFrame::EditInLink returns kODFalse if the part maintaining the link destination could not be found. In this unlikely event, the part should put up a simple alert informing the user that editing the destination of a link is not allowed. An alert for this purpose is included in the Example Part Resources document in the Sample Code folder.

Imaging and Layout

The imaging and layout recipes are listed as follows:

Adding a Display Frame

When a new frame is created, it automatically connects itself to its part as one of its display frames. This ensures that frames are always valid and usable. There is no extra work the object which creates the new frame need do besides create the frame itself.

On the other side of the process, the part being displayed must respond to the call to add the new display frame:

```
void DisplayFrameAdded(in ODFrame* frame)
```

There are probably a number of things the part must do in response to gaining a new display frame. Most of these things will depend on the nature and implementation of the part itself; what these might be are left as an exercise for the reader. However, there are a few general actions a part should take:

- Add the new display frame to the part's list of its display frames. This list, like the part's other internal structures, is completely hidden from OpenDoc. The developer may represent this list any way he chooses. The simplest of parts may be able to do without such a list, but most parts will require it.
- Validate the view type and presentation of the new frame. The part must support the required set of view types. Other kinds of view types or presentations are optional. The part should inspect these values and correct them if necessary. See the recipe for View Types and Presentations for more detail.
- Add part info to the frame. The *partInfo* field of a frame is a convenient place for a part to store information about that view of itself. It can be anything from a simple ID to a pointer to a complicated structure or a helper object. The part info for a frame is stored in the storage unit for the frame, not the part. If the part has many frames of which only a few are internalized, only the part info of those frames will be internalized and take up memory.
- If the frame being added is a root frame, the part may want to activate that frame. See the recipe on Activation for more information on how to do that.
- Do not try to negotiate with the containing part for a different frame shape. The call to CreateFrame has not returned yet, so the containing part has not had a chance to add the new frame to its contents. Unless the containing part is very clever, it will not be able to negotiate a new shape for the frame at this point. It's better to wait until the embedded part's first time receiving a FacetAdded call.

Synchronizing with a Source Frame

Parts may have multiple display frames. Sometimes, two or more views of a part must be synchronized. This is necessary to allow the part to update one frame when editing has taken place in another. In many cases, a part can determine these dependencies internally. But there are some cases when parts need to be synchronized externally.

A typical situation where frames should be synchronized occurs when the user opens a window view of an embedded frame. The embedded part's Open() method is called, wherein it will create the new window. The new window will have a root frame which will be added as a display frame of the part. Now, since the part is in control of this process, it already knows that it must keep these two views synchronized. However, if the part has any embedded parts, they won't know that the new display frames added to them should be synchronized with other frames. So the root part of the new window should use AttachSourceFrame() to do that.

The part being displayed should take whatever action necessary to synchronize the frames. As a minimum, if the two frames are the same kind of presentation, it should duplicate embedded frames in one frame into the other.

Adding and Removing Facets

This recipe contains instructions on how to add and remove facets.

Adding a Facet

Parts are notified when a facet is added to a display frame.

```
void FacetAdded(in ODFacet facet)
```

As when a display frame is added to a part, the part must take actions to handle the addition of a new facet to one of its display frames. Some of these actions are dependent on the nature of the part, but some are fairly standard. These are:

- Create facets for embedded frames. If there are embedded frames that should be visible within the new facet, the part should make them visible using the Making a Frame Visible recipe.
- Add part info to the facet. Part info is stored in facets less often than in frames, but the option is available to help parts distinguish different facets. The part may also use part info to store graphics-system-related information used for displaying that facet. If a part does want to store part info in its facets, it should be added when the facet is added to the part.

```
MyPartFacetAdded(Environment* ev, MyPart* somSelf, ODFacet* facet)
{

 ODFrame* frame = facet->AcquireFrame(ev);
 ODEmbeddedFramesIterator iter = somSelf->
                                 CreateEmbeddedFramesIterator(ev, frame);
 frame->Release(ev);

 for (ODFrame* embedded = iter->First(ev);
   iter->IsNotComplete(ev);
   embedded = iter->Next(ev))
 {
  if (/* Check to see if embedded frame is visible */)
  {

  // Set up new facet geometry...
  childFacet = facet->CreateEmbeddedFacet(ev, embedded, ...);
  // Set up part info...
  childFacet->SetPartInfo(ev, (ODInfoType)partInfo);
  }
 }
 delete iter;
}
```

Removing a Facet

Parts are also notified when a facet is removed from a display frame.

```
void FacetRemoved(in ODFacet facet)
```

The part must handle the removal of facets from its display frames. The actions the part needs to perform to do this will depend on what it did when the facet was added. Typically the part should:

- Remove facets for embedded frames.
- Remove part info from the facet.

```
MyPartFacetRemoved(Environment* ev, MyPart* somSelf, ODFacet* facet)
{
 ODFacetIterator iter = facet->CreateFacetIterator(ev,
            kODChildrenOnly, kODFrontToBack);
 for (ODFacet* childFacet = iter->First(ev);
   iter->IsNotComplete(ev);
   childFacet = iter->Next(ev))
 {
  // delete part info if it was allocated off the heap
```

```
  partInfo = childFacet->GetPartInfo(ev);
  delete partInfo;
  facet->RemoveFacet(ev, childFacet);
 }
 delete iter;
}
```

Adjusting the Active Border

The display of the active frame border is managed by OpenDoc. Neither the active part nor its containing part needs to display the active border; OpenDoc itself does that. All the active part needs to do to get the active border is to acquire the Selection focus, and relinquishing the focus will remove the border.

Even though the containing part doesn't have to display the active border of an embedded frame, it does have to deal with the border to some extent. There are two things the containing part must do:

- Adjust the active border's shape. This amounts to clipping the shape where it is obscured by intrinsic content or embedded frames in the containing part.
- Ensure that the active border is not overwritten by intrinsic content or embedded parts it obscures. This requires that the containing part clip the active border shape from the obscured parts. See the Clipping Embedded Facets recipe for information on how to do that.

Adjusting the Border Shape

When an embedded frame acquires the selection focus, or a frame that already has the focus has its frame shape changed, OpenDoc will calculate a new active border shape, and then ask the frame's containing part to adjust it. OpenDoc will call AdjustBorderShape() on the containing part, passing the shape to be adjusted, and the facet it is associated with.

Adjusting the active border shape is a straightforward process. Just clip out all the portions that are obscured by the containing part's intrinsic content or embedded frames. Do not clip the shape to the containing part's clip shape; OpenDoc will do that.

About Reference Counts

The AdjustBorderShape() method is considered a source for the returned shape. It may be creating a new shape, which will have to be released later. For that reason, if all your part does is return the *shape* parameter, it must make sure to inflate the reference count first by calling its Acquire method. Remember, the *shape* parameter may be released by the calling code after the method exits, so you can't rely on that reference to keep the shape alive.

Clipping Obscured Content and Parts

After the containing part has adjusted the active border shape, it should store the adjusted shape in a variable or field. When the stored shape is non-null, the part should make sure that intrinsic content or embedded frames are clipped so they do not overwrite the active border.

When the active border is moved to a different frame, OpenDoc will notify the containing part by calling AdjustBorderShape with a value of kODNULL for the shape. The containing part can then un-clip obscured content and embedded frames. If the containing part receives several AdjustBorderShape calls in a row that all have non-null shapes, it should use the union of all those shapes for clipping its content and embedded frames. This is because there may be more than one facet on the active frame embedded in the same containing part.

Note that the sample code shown below does not deal with intrinsic content, but only with embedded parts. See the Clipping Embedded Facets recipe for an example of how to generalize this code to deal with intrinsic content

```
#include "Facet.xh"
#include "FacetItr.xh"
#include "Frame.xh"
#include "Shape.xh"
#include "Trnsform.xh"

  . . .

ODShape* CMyPart::AdjustBorderShape(Environment *ev,
  ODFacet* embeddedFacet,
  ODShape* shape)
{
 ODFacet* dispFacet = embeddedFacet->GetContainingFacet(ev);
 ODCanvas* biasCanvas = dispFacet->GetCanvas(ev);

 if (shape == kODNULL)
 {
  fActiveBorderShape->Release(ev);
  fActiveBorderShape = kODNULL;
  this->ClipEmbeddedFacets(ev, dispFacet);
  return kODNULL;
 }

 ODTransform* xform = kODNULL;
 ODShape* tShape = kODNULL;
 ODShape* adjusted = shape->Copy(ev);

 xform = embeddedFacet->AcquireExternalTransform(ev, biasCanvas);
 adjusted->Transform(ev, xform);  // Now in cont frame coords (mine)
 xform->Release(ev)

 ODFrame* dispFrame = dispFacet->AcquireFrame(ev);
 ODShape* dispShape = dispFrame->AcquireUsedShape(ev, biasCanvas);
 adjusted->Intersect(ev, dispShape);
 dispShape->Release(ev);

 ODFacet* facet = kODNULL;
 ODBoolean above = kODFalse;
 ODFacetIterator* facets = dispFacet->CreateFacetIterator(ev,
                                    kODChildrenOnly,
                                    kODBackToFront);
                                    for (facet = facets->First(ev);
   facets->IsNotComplete(ev);
   facet = facets->Next(ev))
 {
  if (above)
  {
   ODFrame* frame = facet->AcquireFrame(ev);
   tShape = ODCopyAndRelease(ev,
            frame->AcquireUsedShape(ev, biasCanvas));
   xform = facet->AcquireExternalTransform(ev, biasCanvas);
   tShape->Transform(ev, xform);
   adjusted->Subtract(ev, tShape);
   tShape->Release(ev);
   xform->Release(ev);
   frame->Release(ev);
  }
  else
  {
   above = (facet == embeddedFacet);
  }
 }

 if (fActiveBorderShape == kODNULL)
  fActiveBorderShape = adjusted->Copy(ev)
 else
  fActiveBorderShape->Union(ev, adjusted);

 xform = embeddedFacet->AcquireExternalTransform(ev, biasCanvas);
 adjusted->InverseTransform(ev, xform);  // Now in embedded frame coords
 xform->Release(ev)

 this->ClipEmbeddedFacets(ev, dispFacet);
 return adjusted;
}
```

Clipping Embedded Facets

Containing parts manage the clipping of their embedded parts. Containing parts use the clip shape of embedded facets to indicate how embedded parts should clip their drawing operations. A facet's clip shape indicates which portion of the facet is visible within its containing facet. If the containing facet is itself obscured, that does not affect the embedded facet's clip shape. To find out what portion of a facet is visible on its canvas, use ODFacet::AcquireAggregateClipShape.

At the risk of sounding repetitious, a containing part needs to update its embedded facets' clip shapes whenever the way they are clipped changes. There are a number of cases where that may happen when:

- A content is rearranged or reformatted
- An embedded part is resized
- The active border shape moves from one frame to another
- Selection handles are shown or hidden
- An embedded frame's UsedShape is changed

If the containing part does not allow embedded frames to be overlapped (like a text part), the number of situations requiring clipping will be minimal. If a containing part allows complex, overlapping arrangements of embedded frames, like a drawing or presentation part, then there will be many situations that require clipping.

Responsibilities

This section discusses the different responsibilities of the containing part and the embedded part.

Containing part
The responsibilities of the containing part are as follows:

- Manage the clip shape of embedded facets. Ensure embedded parts don't overwrite other parts, the containing part's intrinsic content, or the active border shape.
- Display in the portion of an embedded frame not included in the frame's UsedShape. The embedded part had guaranteed that it won't draw outside its display frame's UsedShape, so the containing part has to do it.

Embedded part
The embedded part is expected to do the following:

- Clip drawing operations to display the facet's aggregate clip shape or window aggregate clip shape.
- Never draw outside of the display frame's UsedShape (except root parts).
- If displaying asynchronously, update clipping when GeometryChanged notification indicates that the clip shape has changed.

About the Sample Code

The code below follows a very simple recipe for clipping embedded facets. More sophisticated techniques are possible, and may be required for advanced imaging features such as translucency or compositing.

The basic recipe for clipping an embedded facet is shown below:

- Start with its frame shape.
- Subtract from that the shapes of all the other facets and intrinsic content within the containing part that obscure the frame.

Some tips:

- When a facet is obscured by another facet, use the UsedShape of the obscuring facet's frame to clip the facet.

- Make sure that all intrinsic content is accounted for. This includes selection handles, active frame borders, frame adornments, etc. Anything that isn't an embedded frame but you don't want overwritten by an asynchronously displaying part (such as a clock or movie) must be included.

- If a containing part receives a UsedShapeChanged notification, you only need to re-clip the facets behind the changed one. That recipe is the same as the one below, but just start at the changed facet, not at the uppermost one. Same thing if the active border changes-just re-clip the facets behind the facet of the active frame (or the previously active frame).

CMyPart::GetFacetForEmbeddedContentItem()
Given a content object in the containing part, this method returns the embedded facet that needs to be clipped. If the containing part only allows one facet per embedded frame in each of its display frames, it doesn't need to store a pointer to the facet in the content object. In fact, that would be fairly confusing, and would just end up paralleling the facet list in the frame. Instead, this method will compute the intersection of the set of the immediate children of the containing part's display facet, and the set of the facets of the embedded frame. In that case, the intersection should be a single frame.

If a containing part allows multiple facets on an embedded frame in its content, it will need to keep an explicit reference to a facet in the content object. In that case, this method can just return that reference.

Sample Code

```
#include "Facet.xh"
#include "FacetItr.xh"
#include "Frame.xh"
#include "Shape.xh"
#include "Trnsform.xh"
#include "FocusLib.h"
  . . .

void CMyPart::ClipEmbeddedFacets(Environment* ev, ODFacet* facet)
{
 ODCanvas* biasCanvas = facet->GetCanvas(ev);
 ODGeometryMode geoMode = GetCanvasGeometryMode(ev,
                                                facet->GetCanvas(ev));
 ODShape* workingClip = ODCopyAndRelease(ev,
                                         facet->AcquireClipShape(ev,
                                         biasCanvas));
 workingClip->SetGeometryMode(ev,geoMode);

 // If an embedded frame is active, the active border should obscure
 // other embedded frames.
 // Border was recorded in ::AdjustActiveBorder()
 if (fActiveBorderShape != kODNULL)
  workingClip->Subtract(ev, fActiveBorderShape);

 ODFacet* embFacet = kODNULL;
 ODFrame* embFrame = kODNULL;
 ODShape* newClipShape = kODNULL;
 ODShape* newMaskShape = kODNULL;
 ODTransform* clipTrans = kODNULL;

 // Compute clipping by iterating all content, intrinsic and embedded
 ContentIterator* contents = this->CreateContentIterator(kODFrontToBack);
 for (item = contents->First();
      contents->IsNotComplete();
      item = contents->Next())
 {
  if (item->IsIntrinsic())
  {
   // Clip intrinsic content here if you need to
   // ...

   // Intrinsic content should obscure underlying frames
   // GetMaskShape() also includes selection handles if item is selected
   workingClip->Subtract(ev, item->GetMaskShape());
  }
  else
  {
   embFacet = this->GetFacetForEmbeddedContentItem(facet, item);
   embFrame = embFacet->AcquireFrame(ev);

   // Start with facet's frame shape
```

```
    newClipShape = ODCopyAndRelease(ev, embFrame->
                                    AcquireFrameShape(ev, biasCanvas));

    // Get UsedShape to obscure underlying facets
    newMaskShape = ODCopyAndRelease(ev, embFrame->
                                       AcquireUsedShape(ev, biasCanvas));

    clipTrans = embFacet->AcquireExternalTransform(ev, biasCanvas);

    // Now in containing frame coordinates
    newClipShape->Transform(ev, clipTrans);
    newClipShape->Intersect(ev, workingClip);

    // Now in embedded frame coordinates
    newClipShape->InverseTransform(ev, clipTrans);
    embFacet->ChangeGeometry(ev, newClipShape, kODNULL, biasCanvas);

    // Now in containing frame coordinates
    newMaskShape->Transform(ev, wclipTrans);
    workingClip->Subtract(ev, newMaskShape);

    embFrame->Release(ev);
    newClipShape->Release(ev);
    newMaskShape->Release(ev);
    clipTrans->Release(ev);
   }
  }
  delete contents;

  workingClip->Release(ev);
}

ODFacet* CMyPart::GetFacetForEmbeddedContentItem(ODFacet* facet,
                                      ContentItem* embeddedItem)
{
  // Find the facet that is facet of "embeddedFrame" and contained by "facet"
}
```

Content Update Notification

All OpenDoc parts are responsible for notifying their containing part when their content changes. This notification enables a containing part to update links that include embedded data. This document summarizes the responsibilities of a part that does not support linking. Developers of parts supporting linking are referred to the Linking recipes for more information.

Calling ODFrame::ContentUpdated

Parts call the ContentUpdated methods of their display frames to notify their containing parts of changes to their content. When a change originates in a part, for example, in response to keyboard input, the part should call the ContentUpdated method of each display frame affected by the change, passing it an update ID for the change. For changes originating within a part, this ID should be obtained by calling the UniqueUpdateID method of the session object. For example:

```
displayFrame->ContentUpdated(ev, mySession->UniqueUpdateID(ev));
```

It is not necessary to call ContentUpdated on every individual change, for example, on every keystroke. A part can batch individual changes into one call to ContentUpdated, by waiting for a reasonable pause in changes or until the part loses the selection focus. Be sure to call UniqueUpdateID each time ContentUpdated is called, however.

Implementing EmbeddedFrameUpdated

A part that does not support linking does not need to take any action in its EmbeddedFrameUpdated method. In particular, it does not need to pass the notification to its containing part.

Creating an EmbeddedFramesIterator

Part editors must be able to create iterators for their embedded frames. When a part receives a CreateEmbeddedFramesIterator call, it should create the iterator and return it.

Creating the Iterator

The part editor DLL will include implementations for subclasses of ODPart and ODEmbeddedFramesIterator, and perhaps other extensions as well. To create the embedded frames iterator the part just allocates and initializes the instance.

The function for ODCreateEmbeddedFramesIterator is somewhat awkward to use, due to a missing parameter in the Init call. The iterator should only operate on the embedded frames of a particular display frame of the part. However, the InitEmbeddedFramesIterator call only takes an ODPart parameter, and not an ODFrame parameter. Therefore, the private interface between the iterator and the part that allows the iterator access to the embedded frames must be able to handle the mapping to the particular frame.

For instance, after calling InitEmbeddedFramesIterator, the part could make another call to the iterator to let it know which frame it is operating on. Then the iterator would pass the frame back to the part to get the right embedded frames to iterate. Alternately, the part could just remember the mapping of the iterator to the frame, and the iterator could pass itself instead of the frame.

Caveats

The following caveats apply:

- This code does not contain proper error checking.
- Because the OrderedCollection class is a private utility, the implementation is an exercise left for the reader.

Sample Code

```
MyEmbeddedFramesIterator* CMyPart::CreateEmbeddedFramesIterator
                                  (Environment* ev, ODFrame* frame)
{
   MyEmbeddedFramesIterator* iter = new MyEmbeddedFramesIterator;
   this->MapFrameIterator(iter, frame);
   iter->InitEmbeddedFramesIterator(ev, somSelf);
   return iter;
}

OrderedCollection* CMyPart::GetEmbeddedFrames(Environment* ev,
                                              MyEmbeddedFramesIterator* iter)
{
   ODFrame* frame = this->GetFrameForIter(ev, iter);
   OrderedCollection* frameList =
                     this->GetEmbeddedListOfFrame(ev, frame);
   return frameList;
}
```

```
void MyEmbeddedFramesIterator::InitEmbeddedFramesIterator(
                                Environment* ev, ODPart* part)
{
  fPrivateIter = ((CMyPart*)part)->
                 GetEmbeddedFrames(ev, this)->
                 CreateIterator();
}
```

Determining Print Resolution

When a root part of a window receives a print command, it must set up a print job and cause all the pages to be imaged. A subtlety of creating the print job is determining the right resolution. Some graphics systems are resolution-dependent, and their associated print managers need advance knowledge of the resolution of the print job if it's not to be the default 72 dpi. If the root part or an embedded part needs extra resolution, say if it's a 300 dpi graphic, the root part must create a print job that can handle that resolution. Since the root part can't know or guess what resolution is right for all the embedded parts, it must ask them.

The root part must iterate over each embedded frame that will be printed. It uses the ODPart::GetPrintResolution() call to ask the part of each frame what print resolution it requires, and sets up the job with the maximum value of the set of answers. Of course, the root part must also consider the resolution required by its own frames.

Whether a print job needs to know its resolution, and how to set its resolution, are both graphics system dependent.

On OS/2, the print resolution for a print job cannot be set in programs. However, a root part may try to query the resolution of available printers to find one that is satisfactory.

Embedding a Frame

Containing parts may embed other parts, or rather they may embed frames which display other parts. There are several situations that may cause a containing part to embed another part:

- Data was dragged or pasted into the containing part, and the options on the operation indicated the data should be embedded as a part.
- A part was inserted into the containing part using the **Insert** command.
- A content operation within the containing part caused it to embed another part.
- An embedded part asked the containing part for another frame, possibly for a continuation or alternate view of the same part.

In the first three cases, the containing part itself initiates the actual embedding in response to some other action. In the last case, the embedded part initiates the embedding by calling containingPart->RequestEmbeddedFrame().

In any case, the containing part creates a frame in which to embed a part by asking the draft to create one:

```
newFrame = draft->CreateFrame(ev,
      kODFrameObject,
      containingFrame,
      newShape,
      biasCanvas,
      embeddedPart,
      viewType,
      presentation,
      kODFalse, // IsRoot must be false
      isOverlaid);
```

The new frame returned from ODDraft::CreateFrame() has already been initialized. The frame also made itself a display frame of the embedded part by calling

```
embeddedPart->DisplayFrameAdded(ev, somSelf);
```

This ensures that the new frame can be used as soon as it is returned by the draft. For more about what may have happened during DisplayFrameAdded(), see the recipe for Adding a Display Frame.

Once the new frame is returned, the containing part must store it somewhere in its content. How the part stores its content internally is up to the developer. If the new frame is visible within the containing frame, the containing part must create a facet for it. See the recipe for Making a Frame Visible.

Note that if the embedded part tries to negotiate a different frame shape during its DisplayFrameAdded call, the containing part will get the RequestFrameShape call before the CreateFrame call returns. For this reason, an embedded part should not try to negotiate for a new frame shape in its DisplayFrameAdded method. Instead, it should wait until a facet is added to the frame.

Frame Groups and Sequencing

OpenDoc frames have two properties that can be used to indicate relationships among sibling frames within a containing part. The FrameGroup property indicates that two or more frames should be considered part of a group, and the SequenceNumber property describes the ordering within the group.

The ODFrame function includes the following calls for getting and setting the frame group and sequence number:

```
ODULong ODFrame::GetFrameGroup()
void    ODFrame::SetFrameGroup(in ODULong groupID)
ODULong ODFrame::GetSequenceNumber()
void    ODFrame::SetSequenceNumber(in ODULong sequenceNumber)
```

Containing Part Perspective

A frame's containing part has ownership of its FrameGroup and SequenceNumber; the containing part is the only object that may change those values. Only the containing part may call SetFrameGroup and SetSequenceNumber on an embedded frame.

If a frame is not grouped with any other frames, its FrameGroup and SequenceNumber properties should both be set to zero. This is the default state for a frame when it is created, so the containing part doesn't need to set it to be that way.

When a containing part responds to a RequestEmbeddedFrame call, it should place the new frame in the same group as the base frame, and should give the new frame a sequence number equal to the new size of the group. If the base frame was previously ungrouped (FrameGroup was zero), it should first be given a new group ID and a sequence number of one. Valid sequence numbers are contiguously allocated positive integers. If a frame is removed from the middle of a group, the frames above it should be renumbered so that their sequence numbers are contiguous.

A containing part may have a user interface that allows grouping and sequencing of embedded frames, but whether that is appropriate depends on the part's content model.

Embedded Part Perspective

When an embedded part requests an additional display frame from its containing part (via RequestEmbeddedFrame), the new frame will be in the same frame group as the base frame, and will have the highest sequence number in that group. Newly created frames have a frame group and sequence number of zero; the containing part sets those values shortly after creating the new frame. So the embedded part will receive a DisplayFrameAdded call with the new frame, and shortly after a SequenceChanged call notifying it of the new sequence number.

An embedded part has no control over the sequence ordering of its frames. However, it does have complete control over what content it displays within those frames. So if the part needs to change how its content is displayed in a group of frames, it doesn't need to change their sequence numbers, just how it maps those numbers to content display. But in most cases, the mapping should be pretty straightforward-you do want to respect a user's choice of layout arrangement.

Frame Link Status

Frames have a link status that indicates the frame's participation in links maintained by containing parts. All OpenDoc parts that support embedding are responsible for ensuring that the link status of their embedded frames is set correctly. This document summarizes the part responsibilities for a part that does not support linking. Developers of parts supporting linking are referred to the Linking recipe for more information.

Implementing LinkStatusChanged

All parts that support embedding must implement ODPart::LinkStatusChanged. All they must do is pass the notification along to all embedded frames by calling ODFrame::ChangeLinkStatus. It is not necessary to internalize an embedded frame to make this call; it can be postponed until the embedded frame is internalized for some other reason.

Internalizing an Embedded Frame

It is the responsibility of every part to ensure that the link status of their embedded frames is set correctly. The easiest way for a part editor that does not support linking to do this is to call the frame's ChangeLinkStatus method immediately after internalizing the embedded frame. The ODLinkStatus parameter should be the link status of the part's display frame containing the embedded frame. For example:

```
ODFrame* embeddedFrame = myDraft->AcquireFrame(ev, embeddedFrameID);
embeddedFrame->ChangeLinkStatus(ev, myDisplayFrame->GetLinkStatus(ev));
```

Embedding a Frame

Parts must set the link status of frames they embed, be it by a paste, drop, or some other means. If the part does not support linking, it should call the frame's ChangeLinkStatus method immediately after the frame is embedded and internalized.

Making a Frame Visible

OpenDoc maintains a large amount of information about what parts are visible in a window, so that it may display them and dispatch events to them properly. But OpenDoc knows very little about the embedding structure of a document, as that information is stored by parts in encapsulated internal representations. Therefore, containing parts must tell OpenDoc about embedded frames that are visible. The way to do this is to make a facet for each place in which a frame is visible within a containing part.

An embedded frame may become visible when the containing part has scrolled or moved it into view, when it was just embedded, when an obscuring piece of content was removed, etc. When this happens, the containing part must ensure that there is a facet to display the frame. If there is no facet already, it can make one by asking a facet of its display frame to create it. Note that the containing part must create a facet for each place the frame becomes visible, which means it must create a facet for each facet of the containing part's display frame(s).

The following fragment makes a frame called "embeddedFrame" visible within a display frame called "frame" of a containing part. The clip shape of the new facet is set to be the same as the embeddedFrame's frame shape, but your part should alter that shape to reflect actual clipping. The external transform of the new facet is set from the internal content information stored within the containing part; the function ProxyForFrame() returns the internal object the part uses to hold information about the embedded frame within its content model. The new

facet is created in front of all its sibling facets, if any.

```
ODShape* clip = kODNULL;
ODTransform* xform = kODNULL;
ODFrameFacetIterator* facets = frame->CreateFacetIterator(ev);
for (ODFacet* facet = facets->First(ev);
    facets->IsNotComplete(ev);
    facet = facets->Next(ev))
{
    clip = ODCopyAndRelease(ev, embeddedFrame->AcquireFrameShape(ev));
    xform = ODCopyAndRelease(ev,
                ProxyForFrame(embeddedFrame)->transform);
    facet->CreateEmbeddedFacet(ev, embeddedFrame, clip, xform,
        kODNULL, kODNULL,    // No special canvas or bias
        kODNULL,             // Sibling facet
        kODFrameInFront);    // Frontmost
    clip->Release(ev);
    xform->Release(ev);
}
```

Creating and Using Offscreen Canvases

There are several situations where parts may want to create offscreen canvases. Two common cases are double-buffering and image manipulation. Embedded parts are likely to create offscreen canvases, so they may double-buffer output for greater display efficiency and quality. Containing parts may place embedded parts on offscreen canvases, so they can graphically manipulate the imaging output of the embedded part, and perhaps combine it with their own.

Creating an Offscreen Canvas

To create a canvas, a part must first create a presentation space associated with a memory device context and then call ODFacet::CreatePlatformCanvas to create the platform canvas object. Then it can use the ODFacet::CreateCanvas() factory method to create an actual ODCanvas.

```
// As for your graphics system
PlatformCanvas = facet->CreatePlatformCanvas(ev, hps);

newCanvas = facet->CreateCanvas(ev,
      kGraphicsSystem,                    // Enum code for your g.s.
      platformCanvas,
      kODTrue,                            // Using a dynamic canvas
      kODTrue);                           // This is OFFSCREEN
newCanvas->SetOwner(ev, fSelf);           // fSelf is my PartWrapper
```

Containing part
For a containing part to place an embedded part on an offscreen canvas, it must attach the canvas to the embedded facet. This must be done at facet creation time, before the embedded part gets notified of the new facet via FacetAdded(), where it may attempt to attach its own offscreen canvas (see below).

```
embeddedFacet = facet->CreateEmbeddedFacet(ev,
       frame, clipShape, externalTransform,
       newCanvas,                // Canvas created above
       kODNULL,                  // No biasCanvas
       kODNULL, kODInFront);     // Frontmost facet
```

If an already-existing embedded facet has no canvas of its own, the containing part may add one. In that case, embedded parts will be notified automatically that their display canvas has changed.

```
facet->SetCanvas(ev, newCanvas);
```

Embedded part
An embedded part may attach a canvas to one of its facets at any time. During FacetAdded() is a good time to do this. The part may also choose to move the facet to an offscreen canvas later, for whatever reasons.

```
facet->SetCanvas(ev, newCanvas);
```

If the facet already has a canvas (which must have been assigned by the containing part), the embedded part may not attach a canvas to that facet. If it still wants to image on its own canvas, it must create a subframe of the facet's frame, create a facet on that subframe, and attach the canvas to that facet.

Imaging on an Offscreen Canvas

If your part is written correctly, you shouldn't have to do anything differently to image on an offscreen canvas. Some simple rules to follow:

- Obtain the canvas to image on from the facet passed as a parameter to Part::Draw(). This may be an offscreen canvas.
- If a part needs to draw interactively (for example, for rubber-banding or sweeping out a selection), it can image directly on the root facet's canvas, which should be the same as the window's display environment. All features of the canvas drawing environment computed by the facet are duplicated for the window environment: GetWindowFrameTransform, ...ContentTransform, ...AggregateClipShape. Use these values to image on the window environment.
- If you perform any imaging on your facet's canvas asynchronously (for example, at idle time instead of in response to a ::Draw() call, you must call ODFacet::DrawnIn() to allow proper updating of offscreen canvas contents to onscreen. If a part's facet is moved to an offscreen canvas while it is running, it will receive a ::CanvasChanged() call, which will allow it to transfer its asynchronous imaging to the new canvas.
- Use the Invalidate() and Validate() calls in ODFrame and ODFacet to mark whether areas on a canvas need repainting. These will either make the corresponding calls in the underlying window function, or accumulate the invalid area on an offscreen canvas. This way, invalidation is performed the same way for on- or offscreen canvases.

Updating an Offscreen Canvas

The part which owns an offscreen canvas is responsible for transferring its contents to its parent canvas. This is because only the part which created the canvas can be assumed to know how to transfer its contents. The parent and child canvases may have different formats (for example, bitmap vs. display list), or the owner may want to transform the canvas contents as it copies them (for example, rotate or tint).

When a containing part has placed an embedded facet on an offscreen canvas, it should force the embedded part to draw before the containing part itself draws any of its own contents. This ensures that the contents of the offscreen canvas are up to date, and can safely be combined with the containing part's contents. In the containing part's ::Draw() method:

```
embeddedFacet->Draw(ev, invalidShape);

// In case it has embedded parts
embeddedFacet->DrawChildren(ev, invalidShape);

// Draw some content...
// ...
offscreen = MyExtractBitmap(embeddedFacet->GetCanvas(ev));
MyTransferBitmap(offscreen);
```

If embedded parts display asynchronously, the containing part which owns the canvas on which they image will be notified via the ::CanvasUpdated() call. The part may transfer the content from the offscreen canvas at this time, or it may choose to defer it until later, for reasons of efficiency or minimizing excessive redrawing.

Part Drawing

Part editors must be able to display their parts in response to the ::Draw() call.

```
void Draw(in ODFacet* facet, in ODShape* invalidShape)
```

Draw the part in the given facet. Only the portion in the invalid shape needs to be drawn.

There are several steps a part needs to take to perform the imaging.

1. The part should look at the given facet and its frame. Both the frame and the facet may have some part info that the part has placed there, which the part can use to decide how it will display itself. The frame also has *viewType* and *presentation* fields, which indicate what kind of view of the part should display.

2. The part should examine its canvas to see how it should be imaged. The canvas can be obtained from the facet via ODFacet::GetCanvas(). If the canvas' *isDynamic* flag is kODTrue, the part is imaging onto a dynamic device like a CRT; otherwise, it is imaging to a static device like a printer. The part will probably display its content differently for static and dynamic views. For instance, it should not display scroll bars on a static canvas.

3. The part must make sure the platform graphics system is prepared to draw into the correct context for the facet.

4. Draw the part's contents.

5. Restore the old graphics environment.

Asynchronous drawing

Part editors may sometimes need to display their parts asynchronously, that is, not in response to a ::Draw() call. This process is very similar to the basic drawing recipe, with minor modifications.

1. Determine which of the part's frames should be drawn. A part may have multiple display frames, and more than one may need updating. Parts store their display frames in whatever way they want, so we can't tell you how to find them here.

2. For each frame being displayed, all visible facets must be drawn. ODFrame::CreateFrameFacetIterator() returns an iterator which will list all the facets of a frame. Draw the part's contents in each of these facets, using the recipe above.

3. After drawing in a facet, call ODFacet::DrawnIn() on it to tell it you've drawn in it asynchronously. If the facet is on an offscreen canvas, this lets it get copied into the window.

Printing

When the **Print** command is selected from the Document menu, or when a document is dragged to a printer object on the Desktop, the document shell automatically sends an OD_PRINT event to the HandleEvent method of the owning part of the active window's root frame. If the OD_PRINT event is not handled by the root part, the document shell creates a printing facet for the root frame and starts a print job. Since any part can become a root part (via the View As Window command, or by dragging a frame to the Desktop and double-clicking on it), part handlers should support printing if the default print behavior is not satisfactory (that is, if the page formatting needs to be done or if the part contents more than what is displayed in the frame).

Of course, embedded parts will also be printed as the result of a Print command. But for an embedded part it's a lesser task; the part will be told to open a new frame / facet on a printing canvas, and will then be told to draw itself onto that canvas. If a part wants to display differently on a printout than onscreen, it can check whether the canvas is static. It can also retrieve the platform-dependent print job data from the canvas if it needs more information (for instance, to tell whether it is printing to a PostScript printer.)

Printing support, therefore, turns into two tasks: managing the print job and page layout when one is the root part, and imaging oneself onto a printing canvas. We'll cover these in separate sections.

Managing the Print Job and Page Layout

As with imaging, OpenDoc provides no function for managing print jobs; you must use your operating system's native function to do this. (However, an OpenDoc-based framework will probably provide support for printing; check your local listings.) Once you have set up a print job, you stuff it into a new ODCanvas object and use that canvas as the basis for layout and imaging.

There are two kinds of page layout a part can implement; which one you choose depends on how WYSIWYG your part is. In the first and simpler method, the printed page has the same layout as your onscreen appearance. After you create a printing canvas, you create a new facet on the frame to be printed, attach that facet to the printing canvas, and image the facet (and any embedded facets).

The second method is more complex but allows the printed page to have a different layout than the onscreen representation. To allow a new layout, you must create a new frame for your part on the printing canvas, also adding frames for all your embedded parts. In this process, the embedded parts (if they in turn want to lay themselves out differently on the printout) can use frame negotiation to change their size or position.

The basic steps are:

1. Retrieve any persistent print job settings (such as page setup information).
2. Display the printing dialog to the user.
3. Tell the operating system to start the print job.
4. Create a new static canvas, and stuff references to the print job and its drawing environment into the canvas.
5. Create a new frame or facet (depending on your printing-layout model) for yourself on the printing canvas.
6. Draw the root facet of the printing canvas (and its embedded facets, if any.)
7. If your content doesn't fit on one page, scroll the root facet (by offsetting its external transform and clip shape) and draw the next page; repeat until finished.
8. Close the print job.

Imaging Onto a Printing Canvas

Drawing to a printer is typically almost exactly like drawing to the screen. As usual, your part's Draw method will be called when it's time to draw a facet, and it should use the presentation space obtained from the facet-canvas' PlatformCanvas object to do imaging.

To determine whether the canvas is static (non-interactive), you can use the IsStatic method of the canvas. If the canvas is static, you may not want to draw purely interactive features such as scroll bars. Remember that, while a print job is always static, there can be other kinds of static canvases, such as onscreen page previews; a static canvas is therefore no guarantee that a print job is in progress.

References

The OpenDoc recipe Determining Print Resolution.

Removing an Embedded Frame

In the simplest case, removing an embedded frame is straightforward. Just follow these steps:

- Remove all facets on the embedded frame.
- Call embeddedFrame->Remove(ev). This also calls Release on the frame, so don't use the reference after this call.

However, this simple case does not allow the deletion to operate with Undo. Which brings us to...

Undoable Frame Deletion

If you follow the above recipe, you will destroy the embedding structure for all parts embedded within the frame being deleted. At the very least, this will be a lot of work to Undo; at the worst, you won't be able to reconstruct the hierarchy at all. So you should follow a different recipe if you want to Undo, which you almost certainly do.

- Remove all facets on the embedded frame.
- Set the frame's ContainingFrame to kODNULL. This will close any part window views of this frame, as well as correctly indicate the frame has no containing frame.
- Set the frame's InLimbo flag to kODTrue to indicate that it is not actually embedded anywhere.
- Put the frame aside somewhere, and put a reference to it in the Undo ActionData which you record on the Undo history.
- Remove the frame from your content data structures.

If at some point the user chooses to undo the frame deletion, restoring it can be accomplished as follows:

- Get the frame from the undo limbo where you put it, and insert it back into your content data structures.
- Set the embedded frame's InLimbo flag to kODFalse.
- Set the embedded frame's ContainingFrame to your part's display frame (or the correct one if you have more than one).
- If the frame is visible, create a facet for it.

When the part receives a DeleteActionData call, it should commit the action. If the action involves an embedded frame deletion, the deletion should be committed as follows:

- Check the InLimbo flag of the deleted frame. A value of kODFalse will indicate if a frame that was Cut has been Pasted into another containing frame.
- If frame->IsInLimbo(ev) is kODTrue, call frame->Remove (ev), and toss the reference.
- If frame->IsInLimbo(ev) is kODFalse, call frame->Release(ev), and toss the reference.

RequestEmbeddedFrame

One of the methods which part editors that support embedding may need to override is RequestEmbeddedFrame.

As part of this recipe, the part editor will probably want to put the general functionality of creating a frame for embedding into another method, say CreateEmbeddedFrame, which would be called from RequestEmbeddedFrame. Note that RequestEmbeddedFrame should only be called by an embedded part on its containing part in order to get a sibling frame. That is why this call takes a *baseFrame* parameter. The base frame must already be embedded inside the part receiving the RequestEmbeddedFrame message. The CreateEmbeddedFrame method is used in the Clipboard and Insert... recipes.

Caveats

The following caveats apply:

- This code assumes the containing part is not picky about the requested frame shape and external transform, and always returns the requested embedded frame.
- It does not contain proper error checking.
- It assumes all embedded frames are kept in a single list.

Sample code

```
#define INCL_ODAPI
#define INCL_ODFRAME
#define INCL_ODDRAFT
#define INCL_ODSTORAGEUNIT
#include <os2.h>
  . . .
ODFrame* MyPartRequestEmbeddedFrame(MyPart* somSelf, Environment* ev,
      ODFrame* containingFrame,
      ODFrame* baseFrame,
      ODShape* frameShape,
      ODPart* embedPart,
      ODTypeToken viewType,
      ODTypeToken presentation,
      ODBoolean isOverlaid)
{
 ODFrame* newFrame = somSelf->CreateEmbeddedFrame(ev,
       containingFrame,
       frameShape,
       externalTransform,
       embedPart,
       viewType,
       presentation,
       baseFrame->GetFrameGroup(),
       isOverlaid);
}

// Private implementation method:
ODFrame* MyPartCreateEmbeddedFrame(MyPart* somSelf, Environment* ev,
      ODFrame*          containingFrame,
      ODShape*          frameShape,
      ODTransform*      externalTransform,
      ODPart*           part,
      ODTypeToken       viewType,
      ODTypeToken       presentation,
      ODID              frameGroupID,
      ODBoolean         isOverlaid)
{
 ODFrame* newFrame
  = somSelf->GetStorageUnit(ev)->GetDraft(ev)->CreateFrame(ev,
       kODNULL,             // Default type is persistent
       containingFrame,
       frameShape,
       kODNULL,             // No bias canvas
       part,
       viewType,
       presentation,
       kODFalse,            // Not a subframe
       isOverlaid
 newFrame->SetFrameGroup(ev, frameGroupID);
 newFrame->SetSequenceNumber(ev, somSelf->NextInGroup(ev, frameGroupID);
 // Add frame to list of embedded frames, and keep track of
 // the external transform for that frame
 fMyEmbeddedFrameList->Add(newFrame,externalTransform);

 return newFrame;
}
```

Scrolling

This recipe discusses scrolling within OpenDoc.

Coordinate Systems

There are two coordinate systems a part can use within a display frame. The frame coordinate space describes where the frame itself is. The

content coordinate space describes where the part's contents within that frame is.

By itself, a frame has no way to relate these coordinate spaces to that of any canvas. But a facet on the frame can provide Transform objects that will map coordinates in either the frame or content spaces to the space of the facet's canvas (or window if needed). (See GetContentTransform and GetFrameTransform.)

The frame and content coordinate spaces are related by the frame's internal transform. This transform maps coordinates from the content space into the frame space. A part should change its display frame's internal transform to scroll its contents. Here's how that should be done:

Simple Scrolling

Some parts display only content within their display frames. In this case, the entire area within a frame should scroll. Part handlers must have some way to scroll their content that requires no real estate within the frame-perhaps arrow keys or a hand cursor mode.

1. Determine scroll offset (from arrows, cursor motion, etc.)

2. Alter the display frame's internal transform:

```
xform = displayFrame->AcquireInternalTransform(ev);
xform->MoveBy(ev, scrollOffset);
displayFrame->ChangeInternalTransform(ev, xform);
xform->Release(ev);
```

   Note that in the above example, the part may alter the internal transform directly, without having to make a copy of it. This is because the part is the owner of the internal transform. However, the part must still use ChangeInternalTransform to let the display frame know that the transform has been changed.

Partial Scrolling

Some parts display adornments within their display frames that are not part of their scrollable content. The obvious case is a part with scroll bars-the scroll bars should not move when the content in the frame is scrolled. Parts with such frames should keep a content shape which describes which portion of their frame is actually affected by the internal transform. There is no particular place this shape must be stored, as it is internal to the part's private implementation, but it might be handy to keep it in the *partInfo* of its frame. As with other shapes, this shape should be in the frame coordinate space.

Imaging

A part images in a particular facet of one of its display frames, as specified in Part::Draw(). In the simple case, a part will image everything in that frame using the content coordinate transformation. In the case of partial scrolling, the part should image the non-scrolled portion of its frame using the frame coordinate transform, and only the portion of the frame within the content shape should be imaged using the content transform.

User Interface Interaction

A part responding to mouse clicks, dragging, or other geometry-based interactions must use the internal transform of its frame to transform the relevant points into the content coordinate space. If a part scrolls everything in the frame, all mouse clicks, etc., may be transformed. If the part only scrolls a portion of the frame, it must first hit-test the point with the content shape. Points which fall within the content shape should be transformed by the inverse of the internal transform to convert them into the content coordinate space.

Embedding Support

A part which scrolls only a portion of its frame needs to take extra care if it embeds other parts. The clip shapes of any embedded facets need to be intersected with the frame's content shape so that the embedded frames don't overwrite the non-scrolled portion of the containing frame.

The Other Way

The above method is recommended for most cases. There is a second way to implement scrolling that is more resource consumptive and has worse performance, but it may be useful in unusual circumstances.

The alternative is to implement the scrollable and non-scrollable portions of the frame as two separate frames. Each frame is a display frame of the same part, but the non-scrolling frame is the containing frame of the scrolling frame. The *isSubframe* flag of the scrolling frame must be kODTrue for this to work correctly. As usual, there must be a facet for each frame of the part. When the user mouses in the scroll bar in the parent frame, the part changes the internal transform of the child frame. One advantage of this technique is that no special care must be taken to manage the clip shape of any embedded frames.

View Types and Presentations

OpenDoc parts can display themselves in different ways in different frames, or in different ways in the same frame at different times. The part is in control of its display in its frame(s), but there is a protocol for other parts to suggest or request the part to display itself in a particular way.

There are two axes to how a part displays itself in a frame. One is the type of the view, represented by the *viewType* field in ODFrame objects. The *viewType* field indicates whether a frame shows the part as an icon, thumbnail, full content view, etc. The other is the kind of presentation represented by the *presentation* field in ODFrame objects. The *presentation* field describes which of the various ways a part can present itself is being used in that frame. Examples of presentation kinds are: bar chart, pie chart, tool palette, info dialog.

The view type and presentation are represented as tokenized ISO strings, or type ODTypeToken. Parts are required to support all standard view types, but may define their own set of supported presentation kinds.

The objects which take part in this protocol are ODPart, ODFrame, ODDraft and ODWindowState. The methods involved are:

View Type

```
ODFrame: ODTypeToken GetViewType()
```

Returns the view type of the frame.

```
ODFrame: void SetViewType(in ODTypeToken viewType)
```

Sets the view type of the frame. This should only be called by the frame's part.

```
ODFrame: void ChangeViewType(in ODTypeToken viewType)
```

Changes the view type of the frame. This can be called by any part or other object. This sets the value of the frame's view type, then calls fPart->ViewTypeChanged(this) to notify its part of the change.

```
ODPart: void ViewTypeChanged(in ODFrame frame)
```

The part should change its display in the frame to be of the indicated type. Parts must support all standard view types.

Presentation

```
ODFrame: ODTypeToken GetPresentation()
```

Returns the presentation of the frame.

```
ODFrame: void SetPresentation(in ODTypeToken presentation)
```

Sets the presentation of the frame. This should only be called by the frame's part.

```
ODFrame: void ChangePresentation(in ODTypeToken presentation)
```

Changes the presentation of the frame. This can be called by any object. This sets the value of the frame's presentation, then calls fPart->PresentationChanged(this) to notify its part of the change.

```
ODPart: void PresentationChanged(in ODFrame frame)
```

If the part supports the new presentation kind, it should change its display in the frame to be of that kind. But if the part does not support the new presentation kind, it should instead pick a close match or a good default, and call frame->SetPresentation() to correct the value in the frame.

Frame Creation

```
ODDraft: ODFrame CreateFrame(...)
```

Pass the desired view type and presentation for the new frame. The frame will add itself to the part by calling fPart->AddDisplayFrame(this, sourceFrame). The part may correct the presentation if the specified kind is not supported.

```
ODPart: ODFrame RequestEmbeddedFrame(...)
```

Pass the desired view type and presentation for the new frame. Creates the embedded frame and initializes it as above.

```
ODWindowState: ODWindow CreateWindow(...)
```

Pass the desired view type and presentation for the window's root frame. Creates the window's root frame and initializes it as above.

Connecting Frames to Parts

```
ODPart: void DisplayFrameAdded(in ODFrame frame)
```

The part gets the view type and presentation kind for the new display from the fields in the frame. The part may correct the presentation if the specified kind is not supported.

```
ODPart: void InitPartFromStorage()
```

When the part internalizes its display frame(s), it gets the view type and presentation kind for each display from the fields in the frame. The part may correct the presentation if the specified kind is not supported. This may occur if the part was previously edited by a different editor which supported the same type of part.

User Interface

The user interface recipes are listed as follows:

- Activation
- Basic Event Handling
- Dialogs
- Menus
- Pop-Up Menus
- Opening and Closing Windows
- Opening a Part into a Window
- Window Events
- Properties Notebook
- Views
- Help
- Undo
- Using Resources in OpenDoc
- Facet Windows
- Using Embedded PM Controls within a Facet
- Implementing a Dispatch Module
- Implementing a Focus Module

Several of the code examples in these recipes are taken from the container part sample included in the Toolkit.

The following example is for the HandleEvent:

```
SOM_Scope ODBoolean  SOMLINK ContainerPartHandleEvent(ContainerPart *somSelf,
                                                      Environment *ev,
                                                     ODEventData* event,
                                                     ODFrame* frame,
                                                     ODFacet* facet,
                                                     ODEventInfo* eventInfo)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartHandleEvent");

    ODBoolean handled = kODFalse;

    SOM_TRY
    switch (event->msg)
    {
       ...
       case WM_BUTTON1DOWN:
          {
          ODPoint windowPt(ODIntToFixed(SHORT1FROMMP(event->mp1)),
                           ODIntToFixed(SHORT2FROMMP(event->mp1)));
          handled = somSelf->HandleButton1Down(ev,
                                               facet,
                                               &windowPt,
                                               event);
          break;
          }
       ...
```

```
        case WM_MOUSEMOVE:
           {
           ODPoint windowPt(ODIntToFixed(SHORT1FROMMP(event->mp1)),
                            ODIntToFixed(SHORT2FROMMP(event->mp1)));
           handled = somSelf->HandleMouseMove(ev,
                                              facet,
                                              frame,
                                              &windowPt,
                                              event);
           }
           break;
        ...
        case WM_CHAR:
           if (!((SHORT1FROMMP(event->mp1) & KC_KEYUP)))
           {
             handled = somSelf->HandleKeyDown(ev, frame, event);
           }
           break;

        case WM_ACTIVATE:
           handled = kODTrue; // actually ignored by dispatcher
           if (SHORT1FROMMP(event->mp1) != 0)
              somSelf->ActivatingWindow(ev, frame);
           else
              somSelf->DeActivatingWindow(ev, frame);
           break;
        ...

        case WM_COMMAND:
           if (SHORT1FROMMP(event->mp2) & CMDSRC_MENU ||
               SHORT1FROMMP(event->mp2) & CMDSRC_ACCELERATOR)
           {
             handled = somSelf->HandleMenuEvent(ev, frame, event);
           }
           break;
        ...
        default:
           return kODFalse;
     }
     SOM_CATCH_ALL
     SOM_ENDTRY
     return handled;
}
```

Activation

Foci and the arbitrator
OpenDoc has a flexible and platform-neutral model of part activation, in which part editors request ownership of named foci from an arbitrator. Each session has an arbitrator, which can be obtained from the session using ODSession::GetArbitrator(). The foci are defined as ISO strings (in Foci.idl and its bindings Foci.xh and Foci.h), and the standard set includes:

```
const ODFocusType kODSelectionFocus = "Selection";
const ODFocusType kODMenuFocus = "Menu";
const ODFocusType kODMenuFocus = "Key";
const ODFocusType kODScrollingFocus = "Scrolling";
const ODFocusType kODModalFocus = "Modal";
const ODFocusType kODStatusLine = "StatusLine";
```

These strings can be tokenized using ODSession::Tokenize(). The methods of ODArbitrator and ODFocusSet use the tokenized form of the focus names.

Foci are owned by frames. The active border is drawn around the frame with the selection focus, and Shift-clicks, Alt-clicks and Ctrl-clicks are sent to this frame. Menu and keystroke events are sent to the frames with the menu focus and keystroke focus, respectively. Page Up and Page Down keystrokes are sent to the frame with the scrolling focus. See the Dialogs recipe for information about the modal focus.

There is no assumption that the same frame owns the selection, key and menu foci, even though this will often be the case. Furthermore, OpenDoc does not require that the selection focus be in an active window, even though this is usually what's desired, unless the active window is a modeless dialog. See Requesting a Focus or Focus Set and Window Activation and Frame Activation.

Key Methods

From ODSession:

```
ODArbitrator GetArbitrator();
ODTypeToken Tokenize(in ODType type);
```

From ODArbitrator:

```
ODBoolean RequestFocusSet(in ODFocusSet focusSet,
                          in ODFrame requestingFrame);
ODBoolean RequestFocus(in ODTypeToken focus,
                       in ODFrame requestingFrame);
void RelinquishFocusSet(in ODFocusSet focusSet,
                        in ODFrame relinquishingFrame);
void RelinquishFocus(in ODTypeToken focus,
                     in ODFrame relinquishingFrame);
void TransferFocus(in ODTypeToken focus,
                   in ODFrame transferringFrame,
                   in ODFrame newOwner);
void TransferFocusSet(in ODFocusSet focusSet,
                      in ODFrame transferringFrame,
                      in ODFrame newOwner);
ODFrame AcquireFocusOwner(in ODTypeToken focus);
ODFocusSet CreateFocusSet();
```

From ODPart:

```
ODBoolean BeginRelinquishFocus(in ODTypeToken focus,
                               in ODFrame ownerFrame,
                               in ODFrame proposedFrame);
void CommitRelinquishFocus(in ODTypeToken focus,
                           in ODFrame ownerFrame,
                           in ODFrame proposedFrame);
void AbortRelinquishFocus(in ODTypeToken focus,
                          in ODFrame ownerFrame,
                          in ODFrame proposedFrame);
void FocusAcquired(in ODTypeToken focus,
                   in ODFrame ownerFrame);
void FocusLost(in ODTypeToken focus,
               in ODFrame ownerFrame);
```

Requesting a Focus or Focus Set

A part can request (for one of its frames) ownership of a single focus, or a set of foci (an ODFocusSet) using the ODArbitrator methods RequestFocus and RequestFocusSet(). RequestFocusSet() performs a "two-phase commit". It first asks each current owner if it is willing to relinquish the focus (by calling Part::BeginRelinquishFocus()). If any focus owner is unwilling, the arbitrator aborts the request by calling each part's AbortRelinquishFocus() method, and RequestFocusSet() returns kODFalse. If all focus owners are cooperative, the arbitrator calls each one's CommitRelinquishFocus() method, and RequestFocusSet() returns kODTrue.

Note: OpenDoc does not activate parts. Parts activate themselves by requesting foci for frames.

Transferring a Focus or Focus Set

A focus or set of foci can also be forcibly transferred from one frame to another, without negotiation. ODArbitrator::TransferFocus() and TransferFocusSet() are used for this purpose. When a focus is transferred, the FocusAcquired() method of the new owner's part is called, and

the FocusLost method of the old owner's part is called, unless it is the one doing the transferring (the transferring frame is passed to TransferFocus() and TransferFocusSet()).

Parts should generally request foci rather than transfer them. However, the recipe for handling modal dialogs makes use of TransferFocus. Since modal dialogs might be nested, the recipe consists of saving the current owner of the modal focus, requesting the modal focus, and then transferring it back to the saved owner when the dialog is dismissed.

Note: Actually, if the nested modal dialogs do not themselves contain parts, the above is not strictly necessary, and the part displaying the modal dialog can simply relinquish the modal focus.

While parts generally do not transfer foci, they should be prepared to have certain foci forcibly removed. In this case the FocusLost() method will be called.

Note: FocusAcquired() and FocusLost() are not called during the Request process. The requesting part knows it has acquired the foci because RequestFocus or RequestFocusSet returns kODTrue. The requesting part may be tempted to call its own FocusAcquired() method when RequestFocus() or RequestFocusSet() returns kODTrue, but it is better to use separate private methods which can be called from both FocusAcquired() and when a request returns kODTrue. This way there is a clear separation between the external and internal calls.

Similarly, a part may be tempted to call its own FocusLost() method from its CommitRelinquishFocus() method, but it would be better to keep the external and internal interfaces separate.

Keyboard Navigation

A containing part (such as a forms package) could provide keyboard navigation of embedded parts by transferring the keyboard and/or selection focus from one embedded part to another. The containing part would also have to set the doesPropagateEvents flag of embedded frames, so that it could handle Tab or arrow keys not consumed by the embedded parts. This recipe needs further research, but the containing part should probably request the focus for its own frame (so that an active embedded part can refuse to relinquish it if appropriate), and then transfer the focus to the next embedded frame.

The Scrolling Focus

Page Down and Page Up keys are sent to the part with the scrolling focus. If an embedded part with no scroll bars is active, the user should still be able to page through the document. Typically, containers will have scroll bars, and embedded parts wont, although other configurations are possible. The following rules should be followed:

- An embedded part with the scrolling focus should always relinquish it when asked to.
- A root part in an active window should not relinquish the scrolling focus.

Basic Frame Activation

This section explains the following:

- Foci and focus sets
- Requesting Foci
- Relinquishing Foci

Foci and focus sets
Most frames will require a focus set containing the selection focus, keystroke focus and menu focus when the user clicks in the frame. But a frame with scroll bars might also need the scrolling focus. And a frame for a modeless dialog may require the menu focus, but not the selection focus, so that the active border remains around the frame which opened the dialog, even though it is in an inactive window.

It is best to preserve focus state on a per-frame basis, and this is one reason why those frame objects attached to the part info of ODFrame

objects are so handy.

It is a good idea for each frame object to pre-allocate an ODFocusSet when it is initialized. The ODArbitrator method CreateFocusSet should be used to create the set.

```
SOM_Scope void SOMLINK ContainerPartCommonInitContainerPart(
                       ContainerPart *somSelf, Environment *ev)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartCommonInitContainerPart");

    SOM_TRY
    ...
    _fSelectionFocus = _fSession->Tokenize(ev, kODSelectionFocus);
    _fMenuFocus = _fSession->Tokenize(ev, kODMenuFocus);
    _fKeyFocus = _fSession->Tokenize(ev, kODKeyFocus);
    _fMouseFocus = _fSession->Tokenize(ev, kODMouseFocus);

    _fClipboardFocus = _fSession->Tokenize(ev, kODClipboardFocus);
    if (_fSession->HasExtension(ev, kODExtStatusLine))
    {
      _fStatusLn =(ODStatusLineExtension*) _fSession->
                  AcquireExtension(ev,kODExtStatusLine);
      _fStatusFocus = _fSession->Tokenize(ev, kODStatusLineFocus);
    }
    _fFocusSet = _fSession->GetArbitrator(ev)->CreateFocusSet(ev);
    _fFocusSet->Add(ev, _fSelectionFocus);
    _fFocusSet->Add(ev, _fMenuFocus);
    _fFocusSet->Add(ev, _fKeyFocus);
    ...
    SOM_CATCH_ALL
    SOM_ENDTRY
}
```

Requesting Foci

Frames are activated in a variety of circumstances - on clicks, when a window is first opened, when a window is activated.

Mouse Activation: When a frame is already selected, a mouse event in that frame will go to the containing frame so that it can allow the user to drag the selected frame. The embedded frame will receive a mouse event if the container decides a drag is not happening. Therefore frames should activate themselves if necessary on mouse events.

When the part editor receives an event, and decides it must activate the frame, it requests the set of foci needed by the frame:

```
SOM_Scope void  SOMLINK ContainerPartActivateFrame(
                        ContainerPart *somSelf,
                        Environment *ev,
                        ODFrame* frame)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartActivateFrame");
    SOM_TRY
    if (frame != kODNULL)
       {
       PartInfoRec* pInfo = (PartInfoRec*) frame->GetPartInfo(ev);
       if (!(pInfo->fIsActive))
       {
         ODBoolean succeeded = kODFalse;
         succeeded = _fSession->GetArbitrator(ev)->
                              RequestFocusSet(ev, _fFocusSet,frame);
         if (succeeded)
         {
            somSelf->FocusAcquired(ev, _fSelectionFocus, frame);
            somSelf->FocusAcquired(ev, _fMenuFocus, frame);
            somSelf->FocusAcquired(ev, _fKeyFocus, frame);
```

```
            }
        }
    }
    SOM_CATCH_ALL
    SOM_ENDTRY
}
```

Relinquishing Foci

A part will usually relinquish a focus when another part asks for it via BeginRelinquishFocus(). In this case, the part simply returns kODTrue in BeginRelinquishFocus(), and removes menus, palettes, blinking cursors and so forth in CommitRelinquishFocus().

Most parts will willingly relinquish the common foci when asked, with the exception of the modal focus. The code below is somewhat simplified. The part may in fact wish to relinquish the modal focus to another of its own frames in cases where a nested modal dialog is being displayed from within the frame that owns the modal focus.

If a part does anything more than return kODTrue or kODFalse in BeginRelinquishFocus(), it will have to undo the effects in AbortRelinquishFocus().

A part must also relinquish foci when a frame is going away. In the methods delegated to by Part::DisplayFrameClosed() and Part::DisplayFrameRemoved(), or when the last facet is removed:

```
fArbitrator->RelinquishFocusSet(ev, fFocusSet, frame);
```

A part may also wish to ensure that it keeps various foci together, so it may wish to relinquish a set if it is notified via FocusLost() that it lost a particular focus.

Window Activation and Frame Activation

Not all platforms have a notion of active windows, but for those that do, following these recipes results in sensible behavior in several cases:

- A frame with the selection focus can record this fact when its window is deactivated, and preserve a background selection. When the window is reactivated (for example, by clicking on its title bar), the formerly active frame regains the selection focus. When the user clicks in the content area of an inactive window, the part which receives the event will activate the window, unless the click is in a background selection, in which case the part might allow the selection to be dragged without activating the window. In this case, the destination window will be activated when the drop is completed.
- When a new document is opened from a template, the menus of the root part will appear. The same will be true when the user opens an existing document. It is not necessarily recommended, but an embedded part can choose to record its active state when saved and attempt to restore it when the document is opened. And the part at the root can choose to allow this, or always ensure that it is active itself.

The basic recipe is fairly simple. The frame handler object is attached to the part info in Part::DisplayFrameAdded() and and Part::DisplayFrameConnected(). The frame object attached to the part info makes use of two Boolean properties fNeedsFoci and fHasFoci. The fNeedsFoci flag is set to true if the frame is at the root. The flag is also set to true or false when a window is deactivated, depending on whether or not the frame is active. The fNeedsFoci flag is checked when the part receives a window activate event, and if it is true, the frame is activated.

Creating a New Window

The first time a window, and hence a root frame, is created, the part needs to ensure that the root frame is active when the window becomes

active. The frame handler object will be created and attached in Part::DisplayFrameAdded():

```
SOM_Scope void  SOMLINK ContainerPartDisplayFrameAdded(
                        ContainerPart *somSelf,
                        Environment *ev,
                        ODFrame* frame)

{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartDisplayFrameAdded");
    SOM_TRY
    TempODPart tempPart = frame->AcquirePart(ev);

    if (tempPart == _fPartWrapper)    // frame belongs to me
    {
      ODxOrderedCollectionIterator  displayFramesIter(_fDisplayFrames);
      ODFrame* displayFrame = (ODFrame*) displayFramesIter.First();

      if (displayFramesIter.IsNotComplete() == kODFalse)
        {
        ODxOrderedCollectionIterator  contentsIter(_fContents);
        Proxy*            proxy;
        for (proxy = (Proxy*) contentsIter.First();
            contentsIter.IsNotComplete();
            proxy = (Proxy*) contentsIter.Next())
        {
          proxy->frame->SetContainingFrame(ev, frame);
        }
      }
      PartInfoRec* pInfo = new PartInfoRec;
      pInfo->fGridOn = kODFalse;      //Default is grid off
      pInfo->partwindowID = (ODID)kODNULL;
      if (frame->IsRoot(ev))
        pInfo->fNeedsActivating = kODTrue;

      frame->SetPartInfo(ev, (ODInfoType) pInfo);
      _fDisplayFrames->AddLast(frame);
      frame->Acquire(ev);
      frame->SetDroppable(ev, kODTrue);

      _fNeedToExternalize = kODTrue;

      // if frame view is set do not change it.  If not, make it viewasframe
      if (frame->GetViewType(ev) == kODNullTypeToken)
        frame->SetViewType(ev, _fSession->Tokenize(ev, kODViewAsFrame));
      if (frame->GetPresentation(ev) == kODNullTypeToken)
        frame->SetPresentation(ev, _fSession->Tokenize(ev, kODPresDefault));
    }
    else
    {
       THROW(kODErrInvalidFrame);
    }
    SOM_CATCH_ALL
    SOM_ENDTRY
}
```

Saving and Restoring Windows

When a window is saved in a document, and the document is reopened, the part's DisplayFrameConnected() method is called, and the frame handler object will be attached there.

When a window is activated or deactivated

When a window is first created or is brought to the front, each part will receive an activate event for each facet it has in that window. If the user clicks in the title bar of an inactive window, the window will be brought to the front and each part in the previously active window will get a deactivate event for each facet it has in that window. If a frame has foci, it sets "fNeedsFoci" to true, so that the next time that window is activated, the part can reclaim the foci.

When a different frame in the same window is activated

When the user clicks in a different frame in the same window, one part editor will request foci and install menus and other interface elements. As described earlier, this kind of activation is done in several places (such as mouse events and window activate) , so it is a good idea to record that a frame has the selection focus, so that the method which activates a frame can exit quickly if the frame is already active. After RequestFocusSet() succeeds (such as when handling mouse down and mouse up), fNeedsFoci and fHasFoci are updated.

```
SOM_Scope void  SOMLINK ContainerPartActivateFrame(ContainerPart *somSelf,
                                                    Environment *ev,
                                                   ODFrame* frame)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartActivateFrame");
    SOM_TRY
    if (frame != kODNULL)
       {
       PartInfoRec* pInfo = (PartInfoRec*) frame->GetPartInfo(ev);
       if (!(pInfo->fIsActive))
       {
         ODBoolean succeeded = kODFalse;
         succeeded = _fSession->GetArbitrator(ev)->
                     RequestFocusSet(ev, _fFocusSet,frame);

         if (succeeded)
         {
            somSelf->FocusAcquired(ev, _fSelectionFocus, frame);
            somSelf->FocusAcquired(ev, _fMenuFocus, frame);
            somSelf->FocusAcquired(ev, _fKeyFocus, frame);
         }
       }
    }
    SOM_CATCH_ALL
    SOM_ENDTRY
}
SOM_Scope void  SOMLINK ContainerPartDeActivateFrame(
                        ContainerPart *somSelf,
                        Environment *ev,
                        ODFrame* frame)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart","ContainerPartDeActivateFrame");
    SOM_TRY
    if (frame != kODNULL)
    {
       _fSession->GetArbitrator(ev)->RelinquishFocusSet(ev, _fFocusSet,frame);
       somSelf->FocusLost(ev, _fSelectionFocus, frame);
       somSelf->FocusLost(ev, _fMenuFocus, frame);
       somSelf->FocusLost(ev, _fKeyFocus, frame);
       somSelf->FocusLost(ev, _fStatusFocus,frame);
    }
    SOM_CATCH_ALL
    SOM_ENDTRY
}
SOM_Scope void  SOMLINK ContainerPartActivatingWindow(
                        ContainerPart *somSelf,
                        Environment *ev,
                        ODFrame* frame)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
```

```
                            "ContainerPartActivatingWindow");
    SOM_TRY
    PartInfoRec* pInfo = (PartInfoRec*) frame->GetPartInfo(ev);
    if (pInfo->fNeedsActivating)
    {
       somSelf->ActivateFrame(ev, frame);
       //somSelf->ShowPalette(ev);
       pInfo->fNeedsActivating = kODFalse;
    }
    SOM_CATCH_ALL
    SOM_ENDTRY
}
SOM_Scope void  SOMLINK ContainerPartDeActivatingWindow(
                        ContainerPart *somSelf,
                        Environment *ev,
                        ODFrame* frame)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                            "ContainerPartDeActivatingWindow");
    SOM_TRY
    PartInfoRec* pInfo = (PartInfoRec*) frame->GetPartInfo(ev);
         TempODFrame tempFrame = _fSession->GetArbitrator(ev)->
                                 AcquireFocusOwner(ev, _fSelectionFocus);
    if (frame == tempFrame)
    {
       pInfo->fNeedsActivating = kODTrue;
       if (_fMouseMode != kNormal)
         somSelf->ResetMouseMode(ev);
    }
    else
       pInfo->fNeedsActivating = kODFalse;
    SOM_CATCH_ALL
    SOM_ENDTRY
}
```

Saving An Embedded Selection

If a part does wish to save its selection persistently even when it is embedded, it can do the following.

Note: The Human Interface guidelines do not recommend this. The root part must cooperate by not activating itself if a part already has the selection focus, as shown below.

```
SOM_Scope void  SOMLINK ContainerPartWritePartInfo(
                        ContainerPart *somSelf,
                        Environment *ev,
                        ODInfoType partInfo,
                        ODStorageUnitView* storageUnitView)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                            "ContainerPartWritePartInfo");
    SOM_TRY
    if (partInfo)
    {
      ODStorageUnit* su = storageUnitView->GetStorageUnit(ev);
      ODPropertyName propName = storageUnitView->GetProperty(ev);
      ODSUForceFocus(ev, su, propName, kKindTestContainer);
      ODBoolean needsGrid = ((PartInfoRec*)partInfo)->fGridOn;
      StorageUnitSetValue(su, ev, sizeof(ODBoolean),
                  (ODValue)&needsGrid);
    }
    SOM_CATCH_ALL
    SOM_ENDTRY
}
SOM_Scope ODInfoType SOMLINK ContainerPartReadPartInfo(
                            ContainerPart *somSelf,
```

```
                                Environment *ev,
                                ODFrame* frame,
                                ODStorageUnitView* storageUnitView)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                                "ContainerPartReadPartInfo");
    PartInfoRec* partInfo = kODNULL;
    SOM_TRY
    ODStorageUnit* su = storageUnitView->GetStorageUnit(ev);
    ODPropertyName propName = storageUnitView->GetProperty(ev);
    partInfo = new PartInfoRec;
    ODBoolean needsGrid = kODTrue;

    if (ODSUExistsThenFocus(ev, su, propName, kKindTestContainer))
    {
      StorageUnitGetValue(su, ev, sizeof(ODBoolean),
                    (ODValue)&(needsGrid));
    }
    else
    {
      WARN("Persistent partInfo is missing from this frame. Making one up.");
    }
    partInfo->fGridOn = needsGrid;
    SOM_CATCH_ALL
       ODDeleteObject(partInfo);
    SOM_ENDTRY
    return (ODInfoType) partInfo;
}
```

When root part handles a window activate event:

```
  if (fNeedsFoci)
  {
    ODFrame* selectionFrame
      = fSession->GetArbitrator(ev)->AcquireFocusOwner(ev, fSelectionFocus);
    if (!selectionFrame)
    {
      // activate
    }
    ODReleaseObject(ev, selectionFrame);
}
```

This works because activate events are sent bottom-up to the facets in the window. Note also that the part externalizing the flag must mark the draft as changed when it acquires the selection focus, so that Save is enabled.

Basic Event Handling

This document describes the basics of event distribution to parts. It describes the events a part will need to handle, but does not describe in great detail what should be done with each event. This is heavily content-dependent, and will often be apparent to developers. This section also does not deal with semantic events.

See also:

- Window Events
- Mouse Events
- Menus
- Opening and Closing Windows
- Dialogs
- Drag and Drop

The OpenDoc shell and container applications contain an event loop, and call ODDispatcher::Dispatch() to deliver events to parts. This method takes an event record and returns a Boolean value. If the returned value is kODFalse, the shell may handle the event, otherwise a part, or the dispatcher itself has handled the event.

The dispatcher contains a table of dispatch modules. The dispatcher locates a dispatch module for a given event, and the dispatch module calls one of the following ODPart methods, shown here in IDL:

```
  void Draw(in ODFacet facet,
            in ODShape invalidShape);
  ODBoolean HandleEvent(inout ODEventData event,
            in ODFrame frame,
            in ODFacet facet,
            inout ODEventInfo eventInfo);
```

Standard events
Part editors must handle the following standard events.

- WM_CHAR
- WM_ACTIVATE
- WM_COMMAND
- WM_BUTTON1CLICK
- WM_MOUSEMOVE

Note: Part Editors will need to disable certain portions of their event handling when a draft is read-only. See the Part Init and Externalizing recipe for information on draft permissions.

Standard Events

This section describes the following events:

- Mouse events
- Keyboard events
- Update events
- Activate events
- Disk events
- Menu events

Mouse Events

In addition to the standard mouse events, part editors which support embedding must deal with mouse events in embedded frames which are selected, bundled or viewed as icons or thumbnails. Part editors must also handle special background mouse events to support Drag and Drop. See the Mouse Events and Drag and Drop recipes.

Mouse move events
WM_MOUSEMOVE is passed to your part's HandleEvent along with a frame and a facet. The *where* value is included in the EventInfo structure and contains a point in local (frame) coordinates. The *flags* field in the EventInfo structure tells your part where the mouse click occurred.

```
SOM_Scope ODBoolean  SOMLINK ContainerPartHandleEvent(
                             ContainerPart *somSelf,
                             Environment *ev,
                             ODEventData* event,
                             ODFrame* frame,
                             ODFacet* facet,
                             ODEventInfo* eventInfo)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartHandleEvent");
    ODBoolean handled = kODFalse;
    SOM_TRY
    switch (event->msg)
    {
       case WM_BUTTON1MOTIONSTART:
          if (!eventInfo->flags)
          {
             ODPoint windowPt(ODIntToFixed(SHORT1FROMMP(event->mp1)),
```

```
                        ODIntToFixed(SHORT2FROMMP(event->mp1)));
      handled = somSelf->HandleButton1MotionStart(ev,
                                                  facet,
                                                  &windowPt,
                                                  event);
      break;
   }

case WM_BUTTON1DOWN:
   {
   ODPoint windowPt(ODIntToFixed(SHORT1FROMMP(event->mp1)),
                    ODIntToFixed(SHORT2FROMMP(event->mp1)));
   handled = somSelf->HandleButton1Down(ev,
                                        facet,
                                        &windowPt,
                                        event);
   break;
   }

case WM_BUTTON1CLICK:
 {
   ODPoint windowPt(ODIntToFixed(SHORT1FROMMP(event->mp1)),
                    ODIntToFixed(SHORT2FROMMP(event->mp1)));
   if (eventInfo->flags == kODInEmbedded)
   {
         handled = somSelf->HandleButton1ClickInEmbeddedFrame(ev, facet,
                           eventInfo->embeddedFacet, &windowPt, event);
   }
   else if (eventInfo->flags == kODInBorder)
   {
         handled = somSelf->HandleButton1ClickInBorder(ev, facet,
                           eventInfo->embeddedFacet,
                                     &windowPt,
                                     event);
   }
   else
   {
         handled = somSelf->HandleButton1Click(ev,
                                               facet,
                                               &windowPt,
                                               event);
   }
   break;
 }

case WM_BEGINDRAG:
   {
   ODPoint windowPt(ODIntToFixed(SHORT1FROMMP(event->mp1)),
                    ODIntToFixed(SHORT2FROMMP(event->mp1)));
   return somSelf->HandleMouseDrag(ev, facet,
         (eventInfo->flags == kODInEmbedded ||
                              eventInfo->flags == kODPropagated) ?
                              eventInfo->embeddedFacet:kODNULL,
                              &windowPt, event);
   }

case WM_CONTEXTMENU:
   handled = kODTrue;
   if (_fIgnoreContextMenu && !SHORT2FROMMP(event->mp2))
      break;                // ignore if mouse event
   else
   {
      PartInfoRec* pInfo = (PartInfoRec*) frame->GetPartInfo(ev);
      if (!(pInfo->fIsActive))
            somSelf->ActivateFrame(ev, frame);
      _fPopup->Display(ev);
   }
   break;

case WM_MOUSEMOVE:
   {
   ODPoint windowPt(ODIntToFixed(SHORT1FROMMP(event->mp1)),
                    ODIntToFixed(SHORT2FROMMP(event->mp1)));
   handled = somSelf->HandleMouseMove(ev,
                                      facet,
                                      frame,
                                      &windowPt,
                                      event);
   }
   break;
...
default:
   return kODFalse;
```

```
    }
    SOM_CATCH_ALL
    SOM_ENDTRY
    return handled;
}
SOM_Scope ODBoolean  SOMLINK ContainerPartHandleMouseMove(
                             ContainerPart *somSelf,
                             Environment *ev,
                             ODFacet* facet,
                             ODFrame* frame,
                             ODPoint* where,
                             ODEventData* event)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartHandleMouseMove");
    ODBoolean handled = kODFalse;
    SOM_TRY
    RECTL rclBox;
    PartInfoRec* pInfo = (PartInfoRec*) frame->GetPartInfo(ev);
    switch (_fMouseMode)
    {
                                          // 128682 - faa - starts
       case kNormal:
          if (_fPasteOn)
          {
             if (somSelf->AllowPaste(ev, frame))
             {
                WinSetPointer(HWND_DESKTOP,
                              WinQuerySysPointer(
                              HWND_DESKTOP,
                              SPTR_MOVE, FALSE));
                handled = kODTrue;
             }
             else
             {
                // Been emptied in another docshell
                handled   = kODFalse;
             }
          }
          break;
          // 128682 - faa - ends

       case kRotateSelectCenter:
       case kScaleSelectRefPoint:
          WinSetPointer(HWND_DESKTOP,
                        WinQuerySysPointer(HWND_DESKTOP,
                        SPTR_MOVE,
                        FALSE));
          handled = kODTrue;
          break;

       case kTracking:
          if (_fTrackingFacet)
          {
            if (!SHORT1FROMMP(event->mp2))     // Mouse still captured?
            {
              TempODTransform xform =
               _fTrackingFacet->AcquireWindowFrameTransform(
                               ev, _fTrackingFacet->GetCanvas(ev));
              ODPoint localPoint = *where;   // [137664]
              xform->InvertPoint(ev, &localPoint);      // window to frame coords
              somSelf->UpdateTrackRect(ev,
                                       &localPoint,
                                       _fTrackingFacet,
                                       kUpdateModeContinue);
              _ptEnd = localPoint;
            }
            else                               // mouse is no longer captured
            {
              somSelf->ResetMouseMode(ev);
            }
          }
          else if (facet->GetWindow(ev)->IsActive(ev))
          {
             if (_fSelection->Count() >= 1)
             {
                ODPoint mouse = *where;
                TempODTransform xform =
                        facet->AcquireWindowContentTransform(
                               ev, facet->GetCanvas(ev));
                ODRect bbox;
                xform->InvertPoint(ev, &mouse);
```

```
            }
        }
        break;
    }
    SOM_CATCH_ALL
    SOM_ENDTRY
    return handled;
}
```

Keyboard Events

Keyboard events go to the frame with the keyboard focus, with the exception of the Page Up, Page Down, Home and End keys, which will go to the frame with the scrolling focus, if there is one.

Update Events

Update events are not passed to Part::HandleEvent(). Rather ODPart::Draw() will be called once for each facet in the window. See the Imaging and Layout recipes for examples of drawing.

Activate Events

Activate events are also delivered to each facet in a window, using Part::HandleEvent(). The facets are traversed bottom up (that is, the root facet comes last).

See Window Activation and Frame Activation for example code.

Disk Events

Currently, disk events are not distributed to parts.

Menu Events

OpenDoc passes menu events to the part's HandleEvent method with WM_COMMAND in the message field of the event record. Processing of the WM_COMMAND message in OpenDoc follows the same conventions as in a PM application.

See the Menus recipe for more details.

Special Considerations

Special considerations need to be looked at for:

- Model focus
- Mouse focus
- Propagating events

Modal Focus

Some events are constrained by the modal focus. A mouse click outside the frame with the modal focus will be sent to the modal focus frame, but a click in an embedded frame within the modal focus frame will still go to the embedded frame. When the user clicks outside the modal frame, a facet of kODNULL is passed to HandleEvent(). The part editor should check for this value, and beep or dismiss the dialog as desired.

See the Dialogs recipe for more details about the modal focus.

Mouse Focus

For modal situations, like a polygon drawing tool, the mouse focus captures mouse down, mouse up and mouse moved events. See the Mouse Events recipe.

Propagating Events

This feature of OpenDoc is not likely to be used by most parts. If a containing part sets the "DoesPropagateEvents" property of an embedded frame, the containing part will receive events not handled by the embedded frame, via the HandleEvent() method.

Dialogs

Modal dialogs
In the case of modal dialogs, it is not necessary to create an ODWindow.

As a consequence of OpenDoc's implementation of floating windows, part editors must call ODWindowState::DeactivateFrontWindows() before displaying a modal dialog, and ODWindowState::ActivateFrontWindows() after dismissing it. These methods operate on all floating windows and the foremost non-floating window.

In this case, the part could relinquish the modal focus instead of transferring it back to its previous owner, but by saving and restoring the

owner of the modal focus, this recipe should work for nested modal dialogs, if a dialog was itself built from parts.

A modal dialog box might be implemented by the following code segment:

```
ModalDialogBox(Event)
{
  DosQueryModuleHandle(THISDLL, hmod);
  rc = WinDlgBox(HWND_DESKTOP,event->hwnd,DialogProc,hmod,ID_DIALOG);
}

DialogProc(hwnd,msg,mp1,mp2)
{
  Switch
  {

    Case WM_COMMAND:
          WinDismissDlg(hwnd,TRUE);

  }
}
```

Movable Modal Dialogs

A movable modal dialog allows the user to switch out to other processes. To implement such a dialog, an ODWindow must be created.

Modeless Dialogs

This section explains the following:

- How to show a dialog
- How to close a dialog
- When to activate and deactivate a frame

Showing the Dialog

Suppose the user chooses a menu item to show a modeless dialog. The part editor should do something like the following method of the Clock part, which shows the Alarm Settings dialog:

```
void ClockGlobals::OpenAlarmSettingsDialog(
     Environment* ev, ODFrame* sourceFrame)
{
 ODPart* sourcePart = sourceFrame->AcquirePart(ev);
 ODWindow* window = fSession->GetWindowState(ev)->AcquireWindow(
                              ev, fAlarmSettingsWindow);
 if (window)
 {
  window->Show(ev);
  window->Select(ev);
  ODReleaseObject(ev, window);
 }
 else
 {
  this->CreateAlarmSettingsDialog(ev, sourceFrame);
  this->OpenAlarmSettingsDialog(ev, sourceFrame); // Ooh. Sneaky
 }
 ODReleaseObject(ev, sourcePart);
```

```
}
void ClockGlobals::CreateAlarmSettingsDialog(
Environment* ev, ODFrame* sourceFrame)
{
 ODSLong savedRefNum;
 ODWindow* settingsWindow = kODNULL;
 ODPart* sourcePart = sourceFrame->AcquirePart(ev);

 DosQueryModuleHandle(savedRefNum);
 fAlarmSettingsDialog = WinLoadDlg(kClock_AlarmSettingsDialogID,
                                   (Ptr)SOMMalloc(sizeof(DialogRecord)),
                                   (WindowPtr)-1L);
 EndUsingLibraryResources(savedRefNum);

 settingsWindow = fSession->GetWindowState(ev)->
  RegisterWindow(ev, fAlarmSettingsDialog,
  kODNonPersistentFrameObject,
   kODFalse,    // Keeps draft open
   kODFalse,    // Is resizable
   kODFalse,    // Is floating
   kODFalse,    // do not save
   kODFalse,    // should dispose
   sourcePart,
   fFrameView,  // View Type
   fAlarmSettingsPresentation,  // Presentation
   sourceFrame);

 if (fAlarmSettingsDialog && settingsWindow)
 {
  settingsWindow->Open(ev);
  fAlarmSettingsWindow = settingsWindow->GetID(ev);
  ODReleaseObject(ev, settingsWindow);
 }
 ODReleaseObject(ev, sourcePart);
}
```

Dismissing the Dialog is done when your part receives the WM_COMMAND message during the Override of HandleEvent(). Your part should call WinDismissDlg() to remove the dialog from the screen. Your part should call WinDestroyWindow() when Dialog Window is no longer needed.

Closing a Dialog

When the user clicks in the close box of the modeless dialog, the part may wish to hide the window rather than close it, so that it is not destroyed and can be redisplayed rapidly.

In Part::HandleEvent:

```
 case kODEvtWindow:
 {
  switch (event->message)
  {
       case inGoAway:
   wasHandled = _fClockPart->CloseWindow(ev, frame);
   break;
```

When Activating or Deactivating a Frame

Parts should hide their palettes and dialogs when deactivated, and also when the process is deactivated . In the Test Clock example, the dialogs are shared between part instances.

When the selection focus is lost or gained, the global object managing the dialogs is notified:

```
void ClockFrame::LostSelectionFocus(Environment* ev,
```

```
     ODFrame* proposedFrame)
{
 ODBoolean samePart =
  (proposedFrame && (proposedFrame->
                           GetPresentation(ev) ==
                           fClockPart->fTimePresentation));
 if (!samePart)
  gClockGlobals->SuspendWindows(ev, kODFalse);
}
void ClockFrame::AcquiredSelectionFocus(Environment* ev)
{
 gClockGlobals->AcquiringFocus(ev, fFrame);
 gClockGlobals->ResumeWindows(ev, kODFalse);
}
```

The global object delegates to methods of the ClockFrame class stored in the part info:

```
void ClockGlobals::SuspendWindows(Environment* ev,
     ODBoolean processChange)
{
 OrderedCollectionIterator iter(fDialogFrames);

 for (ODFrame* aFrame = (ODFrame*)iter.First();
   iter.IsNotComplete();
   aFrame = (ODFrame*)iter.Next())
 {
  ClockFrame* clockFrame = (ClockFrame*) aFrame->GetPartInfo(ev);
  if (processChange)
   clockFrame->SuspendProcess(ev);
  else
   clockFrame->SuspendFocus(ev);
 }
}
void ClockGlobals::ResumeWindows(Environment* ev,
ODBoolean processChange)
{
 OrderedCollectionIterator iter(fDialogFrames);

 for (ODFrame* aFrame = (ODFrame*)iter.First();
    iter.IsNotComplete();
    aFrame = (ODFrame*)iter.Next())
 {
  ClockFrame* clockFrame = (ClockFrame*) aFrame->GetPartInfo(ev);
  if (processChange)
   clockFrame->ResumeProcess(ev);
  else
   clockFrame->ResumeFocus(ev);
 }
}
```

When a suspend or resume event is received, the frame objects are notified directly:

```
 case kSuspendResumeMessage:
  {
   ClockFrame* clockFrame = (ClockFrame*) frame->GetPartInfo(ev);
   const short kResumeMask = 0x01; // High byte suspend/resume event
   ODBoolean goingToBackground = (event->message & kResumeMask) == 0;

   if (goingToBackground)
   {
    clockFrame->SuspendProcess(ev);
   }
   else
   {
    clockFrame->ResumeProcess(ev);
   }
  }
```

The frame objects hide and show their windows as needed:

```
void ClockFrame::SuspendFocus(Environment* ev)
{
```

```
 if (fFrame->IsRoot(ev) && fShouldHideOnSuspend)
 {
  TempODWindow window = fFrame->AcquireWindow(ev);
  if (window->IsShown(ev))
  {
   window->Hide(ev);
   fShowWindowOnFocus = kODTrue;
  }
  else
  {
   fShowWindowOnFocus = kODFalse;
  }
 }
}

void ClockFrame::SuspendProcess(Environment* ev)
{
 fInBackground = kODTrue;
 if (fFrame->IsRoot(ev) && fShouldHideOnSuspend)
 {
  TempODWindow window = fFrame->AcquireWindow(ev);
  if (window->IsShown(ev))
  {
   fShowWindowOnResume = kODTrue;
   window->Hide(ev);
  }
  else
  {
   fShowWindowOnResume = kODFalse;
  }
 }
}

void ClockFrame::ResumeFocus(Environment* ev)
{
 if (fFrame->IsRoot(ev) && fShouldHideOnSuspend)
 {
  TempODWindow window = fFrame->AcquireWindow(ev);
  if (fShowWindowOnFocus && !fInBackground)
  {
   fShowWindowOnFocus = kODFalse;
   window->Show(ev);
  }
 }
}

void ClockFrame::ResumeProcess(Environment* ev)
{
 fInBackground = kODFalse;
 if (fFrame->IsRoot(ev) && fShouldHideOnSuspend)
 {
  TempODWindow window = fFrame->AcquireWindow(ev);
  if (fShowWindowOnResume)
  {
   // It may be hidden by user before next Suspend
   fShowWindowOnResume = kODFalse;
   fShowWindowOnFocus = kODFalse;
   window->Show(ev);
  }
 }
}
```

Menus

When a part is initialized (in ODPart::InitPart and ODPart::InitPartFromStorage is should call ODWindowState::CopyBaseMenuBar to obtain its own menu bar object. It can then add its own menus and/or menu items to the menu bar. In the below example ContainerPart::CommonInit is called from both InitPart and InitPartFromStorage.

Getting the Menu Bar

```
SOM_Scope void SOMLINK
ContainerPartCommonInitContainerPart(ContainerPart *somSelf,
                                     Environment *ev)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartCommonInitContainerPart");

    .......(other initialization code).......

    // Since the object is the part's copy,
    // we can add and subtract menus and
    // items without affecting other running parts.
     _fMenuBar = _fSession->GetWindowState(ev)->CopyBaseMenuBar(ev);
    if (_fMenuBar)
    {
       somSelf->InstallMenus(ev);
    }
}
```

Adding Part Menus and Menu Items to the Base Menu Bar

```
SOM_Scope void SOMLINK ContainerPartInstallMenus(ContainerPart *somSelf,
                                                 Environment *ev)
{
  ContainerPartData *somThis = ContainerPartGetData(somSelf);
  ContainerPartMethodDebug("ContainerPart",
                           "ContainerPartInstallMenus");

  HMODULE hmod = NULLHANDLE;
  HWND hwndMenu = NULLHANDLE;
  HWND hwndMoveSubmenu = NULLHANDLE;
  ODPlatformMenuItem mi;
  char string[256];

  //load up our background color menu
  hwndMenu = WinLoadMenu(HWND_OBJECT, _hmod, RESID_BACKGROUNDMENU);

  _fMenuBar->AddMenuLast(ev, RESID_BACKGROUNDMENU, hwndMenu, somSelf);
  _fMenuBar->SetMenuItemText(ev, RESID_BACKGROUNDMENU, kODNULL,
                             "Background");

  //add a submenu to  EDIT->MOVE
   hwndMoveSubmenu = WinLoadMenu(HWND_OBJECT,_hmod,RESID_ARRANGE);
   _fMenuBar->InsertSubmenu(ev, IDMS_EDIT, EDIT_MOVE, hwndMoveSubmenu);

  //add a menu item to the VIEW menu
    mi.id = IDMA_TOGGLE_GRID;
    mi.afStyle = MIS_TEXT;
    _fMenuBar->AddMenuItemLast(ev,IDMS_VIEW,kODNULL, &mi);
    _fMenuBar->SetMenuItemText(ev,
                               IDMS_VIEW,
                               IDMA_TOGGLE_GRID,
                               "Grid On");

}
```

When a part activates itself, it should request the menu focus (along with other foci). Once the part has the menu focus, it should call ODMenuBar:: Display to make its menu the active menu.

Displaying the Menu Bar

```
SOM_Scope void  SOMLINK
ContainerPartFocusAcquired(ContainerPart *somSelf,
                           Environment *ev,
                           ODTypeToken focus,
                           ODFrame* ownerFrame)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartFocusAcquired");

    ......(code for other focus types goes here)......

    if (focus == _fMenuFocus)
         _fMenuBar->Display(ev);
}
```

When a user clicks anywhere within the menu bar, Opendoc will determine which, part has the menu focus and calls Part::AdjustMenus. Here the part can enable, disable, check or uncheck menu items using the methods of ODMenuBar class.

Adjusting the Menu Bar

```
SOM_Scope void  SOMLINK ContainerPartAdjustMenus(ContainerPart *somSelf,
                                                 Environment *ev,
                                                ODFrame* frame)
{
  ContainerPartData *somThis = ContainerPartGetData(somSelf);
  ContainerPartMethodDebug("ContainerPart","ContainerPartAdjustMenus");

  PartInfoRec* pInfo = (PartInfoRec *) frame->GetPartInfo(ev);

  //enable the VIEW->Show as
  _fMenuBar->EnableMenuItem(ev,IDMS_VIEW,VIEW_SHOWAS,kODTrue);

  //put a check next to our default view type
  _fMenuBar->CheckMenuItem(ev,IDMS_VIEW,VIEW_SAFRAME,kODTrue);

  //enable the view properties menu item
  _fMenuBar->EnableMenuItem(ev,IDMS_VIEW, VIEW_PROPERTIES, kODTrue);

  if (pInfo->fGridOn)
   _fMenuBar->SetMenuItemText(ev,IDMS_VIEW,IDMA_TOGGLE_GRID,"Grid Off");
  else
   _fMenuBar->SetMenuItemText(ev,IDMS_VIEW,IDMA_TOGGLE_GRID,"Grid On");

 //enable menu items based on single/multiple/no selection
 switch (_fSelection->Count())
 {
    case 0:     // no selection
      _fMenuBar->EnableMenuItem(ev,IDMS_EDIT, EDIT_SELECTALL, kODTrue);
    break;

    case 1:     // single selection
      _fMenuBar->EnableMenuItem(ev,IDMS_EDIT, EDIT_DELETE, kODTrue);
      _fMenuBar->EnableMenuItem(ev,IDMS_EDIT, EDIT_DESELECTALL, kODTrue);
      _fMenuBar->EnableMenuItem(ev,IDMS_EDIT, IDMA_ROTATE, kODTrue);
    break;

    default: // multiple selection
      _fMenuBar->EnableMenuItem(ev,IDMS_EDIT, EDIT_DELETE, kODTrue);
      _fMenuBar->EnableMenuItem(ev,IDMS_EDIT, EDIT_DESELECTALL, kODTrue);
  };
}
```

Menu events are passed by Opendoc to the Part::HandleEvent method. In this example, menu events are handled by a part-defined method called Part::HandleMenuEvent.

Handling Menu Events

Menu events are generated when the user either clicks on a menu item or uses an accelerator key. Your part's HandleEvent method will be called with a WM_COMMAND event message.

```
SOM_Scope ODBoolean  SOMLINK
ContainerPartHandleEvent(ContainerPart *somSelf, Environment *ev,
                         ODEventData* event, ODFrame* frame,
                         ODFacet* facet, ODEventInfo* eventInfo)
{
 ContainerPartData *somThis = ContainerPartGetData(somSelf);
 ContainerPartMethodDebug("ContainerPart","ContainerPartHandleEvent");

 ODBoolean handled = kODFalse;

 switch (event->msg)
 {
    ....(other messages go here).......

    case WM_COMMAND:
       if (SHORT1FROMMP(event->mp2) & CMDSRC_MENU ||
           SHORT1FROMMP(event->mp2) & CMDSRC_ACCELERATOR)
         handled = somSelf->HandleMenuEvent(ev, frame, event);
       break;
    default:
       return kODFalse;
 }
 return handled;
}


ContainerPartHandleMenuEvent(ContainerPart *somSelf, Environment *ev,
                             ODFrame* focusFrame, ODEventData* event)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartHandleMenuEvent");
    PartInfoRec* pInfo = (PartInfoRec*) focusFrame->GetPartInfo(ev);

    ODBoolean handled = kODFalse;
    ODCommandID command = LONGFROMMP(event->mp1);

    switch (command)
    {
       case IDMA_COLOR_GRAY      :
       case IDMA_COLOR_RED       :
       case IDMA_COLOR_GREEN     :
       case IDMA_COLOR_YELLOW    :
       case IDMA_COLOR_BLUE      :
       case IDMA_COLOR_MAGENTA   :
       case IDMA_COLOR_CYAN      :
       case IDMA_COLOR_WHITE     :
               somSelf->HandleColorMenu(ev, focusFrame, command);
               handled = kODTrue;
               break;
       case EDIT_DELETE     :
               somSelf->DoClear(ev, focusFrame);
               handled = kODTrue;
               break;
       case EDIT_CUT        :
               somSelf->DoCut(ev, focusFrame);
               handled = kODTrue;
               break;
       case EDIT_COPY       :
               somSelf->DoCopy(ev, focusFrame);
               handled = kODTrue;
               break;
       case EDIT_PASTE      :
               somSelf->DoPaste(ev, focusFrame);
               handled = kODTrue;
               break;
```

```
        case EDIT_SELECTALL      :
                somSelf->DoSelectAll(ev, focusFrame);
                handled = kODTrue;
                break;
        case EDIT_DESELECTALL      :
                somSelf->DoDeSelectAll(ev, focusFrame);
                handled = kODTrue;
                break;
        case IDMA_MOVETOFRONT :
                somSelf->MoveToFront(ev, focusFrame);
                handled = kODTrue;
                break;
        case IDMA_MOVETOBACK  :
                somSelf->MoveToBack(ev, focusFrame);
                handled = kODTrue;
                break;
        default:
           break;
    }
    return handled;
}
```

Opendoc provides a default accelerator table for the base menu bar items. Parts can add their part-defined accelerators to this table during menu bar construction in Part::InstallMenus.

Adding Accelerators

If your part has accelerators to add to the base accelerator table, it should add these accelerators during its menu construction.

```
SOM_Scope void SOMLINK ContainerPartInstallMenus(ContainerPart *somSelf,
                                                 Environment *ev)
{
  ContainerPartData *somThis = ContainerPartGetData(somSelf);
  ContainerPartMethodDebug("ContainerPart","ContainerPartInstallMenus");

    //......other menu construction code goes here.............

    memset((PCH)&PartAccel, 0, sizeof(ODACCEL));
    PartAccel.aAccel[0].fs  = AF_SHIFT | AF_VIRTUALKEY;
    PartAccel.aAccel[0].key = VK_UP;
    PartAccel.aAccel[0].cmd = IDMA_MOVETOFRONT;
    PartAccel.aAccel[1].fs  = AF_SHIFT | AF_VIRTUALKEY;
    PartAccel.aAccel[1].key = VK_DOWN;
    PartAccel.aAccel[1].cmd = IDMA_MOVETOBACK;
    PartAccel.aAccel[2].fs  = AF_ALT | AF_VIRTUALKEY;
    PartAccel.aAccel[2].key = VK_UP;
    PartAccel.aAccel[2].cmd = IDMA_MOVEFORWARD;
    PartAccel.aAccel[3].fs  = AF_ALT | AF_VIRTUALKEY;
    PartAccel.aAccel[3].key = VK_DOWN;
    PartAccel.aAccel[3].cmd = IDMA_MOVEBACKWARD;

    _fMenuBar->AddToAccelTable(ev, 4L, &PartAccel);

}
```

Opendoc provides default status line text for the base menu bar items. Parts can add their part-defined status line strings during menu bar construction in Part::InstallMenus.

Adding Status Line Text for Part Menu Items

Status line text for menu items is added when the menu items are added during menu construction.

```
SOM_Scope void SOMLINK ContainerPartInstallMenus(ContainerPart *somSelf,
                                                 Environment *ev)
{
  ContainerPartData *somThis = ContainerPartGetData(somSelf);
```

```
  ContainerPartMethodDebug("ContainerPart","ContainerPartInstallMenus");

    //......other menu construction code goes here.............


   //now lets add status line text strings for our color menu items

     int id []= { IDMA_COLOR_BASE   ,
                  IDMA_COLOR_GRAY   , IDMA_COLOR_BLUE   ,
                  IDMA_COLOR_RED    , IDMA_COLOR_MAGENTA,
                  IDMA_COLOR_GREEN  , IDMA_COLOR_CYAN   ,
                  IDMA_COLOR_YELLOW , IDMA_COLOR_WHITE  };

     int *p = id;
     while(*p) {
      rc = WinLoadString((HAB)0, _hmod, (*p), 255, string);
      if (rc)
         _fMenuBar->SetMenuItemStatusText(ev,*p,string);
      (*p++);
     } // endwhile

}
```

Pop-Up Menus

When a part is initialized (in ODPart::InitPart and ODPart::InitPartFromStorage is should call ODWindowState::CopyBasePopup to obtain its own pop-up menu object. It can then add its own menus and/or menu items to the pop-up. In the below example ContainerPart::CommonInit is called from both InitPart and InitPartFromStorage.

Getting the Pop-Up

```
SOM_Scope void SOMLINK
ContainerPartCommonInitContainerPart(ContainerPart *somSelf,
                                     Environment *ev)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartCommonInitContainerPart");

    ......(other initialization code).......

    // Since the object is the part's copy, we can add and subtract menus and
    // items without affecting other running parts.

     _fMenuBar = _fSession->GetWindowState(ev)->CopyBaseMenuBar(ev);

    if (_fMenuBar) {
       _fPopup = _fSession->GetWindowState(ev)->CopyBasePopup(ev);
       somSelf->InstallMenus(ev);
    }
}
```

Adding to the Pop-Up Menu

```
SOM_Scope void  SOMLINK ContainerPartInstallMenus(ContainerPart *somSelf,
                                                  Environment *ev)
{
  ContainerPartData *somThis = ContainerPartGetData(somSelf);
  ContainerPartMethodDebug("ContainerPart","ContainerPartInstallMenus");

  ODPlatformMenuItem mi;

  ......(menu bar, accelerator, status line code go here).....

  //insert the print item on the pop-up menu
  //Items added to the pop-up menu in this method will always appear
  //on the pop-up

  memset((PCH)&mi, 0, sizeof(MENUITEM));
  mi.id = IDMA_PRINT;
  mi.afStyle = MIS_TEXT;
  _fPopup->AddMenuItemLast(ev, kODNULL, kODNULL, &mi);
  _fPopup->SetMenuItemText(ev, kODNULL, IDMA_PRINT, "Print part");
}
```

When a user presses mouse button 2 to bring up a pop-up menu, the WM_CONTEXTMENU message is passed to the part under the mouse if that part is not in an icon or thumbnail view, and is not bundled. When the part receives this message in its Part::HandleEvent method, it should activate itself if it is not active and display its pop-up menu.

Displaying the Pop-Up Menu

```
SOM_Scope ODBoolean  SOMLINK ContainerPartHandleEvent(ContainerPart *somSelf,
                        Environment *ev,
                        ODEventData* event, ODFrame* frame,
                        ODFacet* facet, ODEventInfo* eventInfo)
{
  ContainerPartData *somThis = ContainerPartGetData(somSelf);
  ContainerPartMethodDebug("ContainerPart","ContainerPartHandleEvent");

  ODBoolean handled = kODFalse;

  switch (event->msg)
  {
   .....(other messages go here).......

     case WM_CONTEXTMENU:
        handled = kODTrue;
        {
         PartInfoRec* pInfo = (PartInfoRec*) frame->GetPartInfo(ev);
         if (!(pInfo->fIsActive)){
               somSelf->ActivateFrame(ev, frame);
         somSelf->AdjustPopupMenu(ev, frame);
         _fPopup->Display(ev);
        }
      break;
  }
}
```

It is up to the part to check and adjust its pop-up menu before displaying it. OpenDoc will not call the part's Part::Adjustmenus; that method is only valid for menu bar menus. There are a number of ways a part can adjust its pop-up menu before displaying it. Below are examples of how ContainerPart adjusts its pop-up menu using the methods ContainerPart::AdjustPopupMenu and ContainerPart::ClearPopupMenu.

Adjusting the Pop-Up Menu

```
SOM_Scope void  SOMLINK ContainerPartAdjustPopupMenu(
                                                ContainerPart *somSelf,
                                                Environment *ev,
                                                ODFrame* frame)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartAdjustPopupMenu");

    //this is a part-defined method that dynamically adds to the pop-up
    //menu with content-editing menu items based on selection

    //call this method to put the pop-up menu into a default state again
    somSelf->ClearPopupMenu(ev, frame);
    //add menu items based on single/multiple/no selection
    switch (_fSelection->Count())
    {
       case 0:     // no selection
        _fPopup->AddDefaultMenuItemBefore(ev, EDIT_SELECTALL, IDMS_HELP);
          break;

 case 1:     // single selection
     {
     _fPopup->AddDefaultMenuItemBefore(ev,EDIT_DELETE,IDMS_HELP);
     _fPopup->AddDefaultMenuItemBefore(ev,EDIT_DESELECTALL,EDIT_DELETE);
     memset((PCH)&mi, 0, sizeof(MENUITEM));
     mi.id = IDMA_SCALE;
     mi.afStyle = MIS_TEXT;
     _fPopup->AddMenuItemLast(ev, kODNULL, kODNULL, &mi);
     _fPopup->SetMenuItemText(ev, kODNULL, IDMA_SCALE, "Scale");
     };
  break;

  default: // multiple selection
    {
    _fPopup->AddDefaultMenuItemBefore(ev, EDIT_DELETE,IDMS_HELP);
    _fPopup->AddDefaultMenuItemBefore(ev, EDIT_DESELECTALL,EDIT_DELETE);
    };
 };
}

SOM_Scope void SOMLINK
ContainerPartClearPopupMenu(ContainerPart *somSelf,Environment *ev,
                            ODFrame* frame)
{
  ContainerPartData *somThis = ContainerPartGetData(somSelf);
  ContainerPartMethodDebug("ContainerPart",
                           "ContainerPartRestoreSelectedMenu");

// check the pop-up for those items dependent on selection that this part
// adds and remove them from the pop-up to return it to a default state

  if(_fPopup->Exists(ev, kODNULL, EDIT_DELETE))
    _fPopup->RemoveMenuItem(ev, kODNULL, EDIT_DELETE);

  if(_fPopup->Exists(ev, kODNULL, EDIT_SELECTALL);
    _fPopup->RemoveMenuItem(ev, kODNULL, EDIT_SELECTALL);

  if(_fPopup->Exists(ev, kODNULL, EDIT_DESELECTALL);
    _fPopup->RemoveMenuItem(ev, kODNULL, EDIT_DESELECTALL);

  if(_fPopup->Exists(ev, kODNULL, IDMA_SCALEL);
    _fPopup->RemoveMenuItem(ev, kODNULL, IDMA_SCALE);
}
```

Opening and Closing Windows

Part editors are required to create an ODWindow instance for each modeless platform window they create. An ODWindow contains a root frame, and thus provides the root of the layout hierarchy. This provides a means for OpenDoc to distribute events to parts, since the main event loop is within the OpenDoc shell.

ODWindow is a fairly thin cover for a platform window pointer. For some operations, the part editor will have to retrieve the platform window

pointer from the ODWindow, and use platform facilities.

This document describes how to create, register, open and close windows. See also the Opening a Part into a Window, Window Events, Dialogs, and Activation recipes.

Creating a Window

Part editors create windows using the standard facilities of the platform. On the OS/2 platform, WinCreateWindow or WinCreateStdWindow calls are used.

Note: Part editors should pass kODFalse for the *isVisible* parameter. Once the window is registered, the Show() method will be called to show it. This is a consequence of the implementation of floating windows in OpenDoc.

Registering a Window

An ODWindow instance is created by registering a platform window with the ODWindowState object, which is obtained from the session like other session-wide services:

```
fWindowState = fSession->GetWindowState(ev);
```

There are two similar methods for registering a window. RegisterWindow() creates and returns an ODWindow containing a newly-created root frame belonging to the supplied root part.

```
ODWindow* window = fWindowState->RegisterWindow(ev,
        platformWindow, // The platform window pointer
        kODFrameObject, // frame type - kODFrameObject or
                        // kODNonPersistentFrameObject
        kODTrue,        // is root window - (that is, keeps the document open)
        kODFalse,       // is resizable - OpenDoc will draw the resize icon
        kODFalse,       // is floating
        kODTrue,        // should save - saved as part of the document
        kODFalse,       // should dispose - part will dispose platform window
        fPartWrapper,   // root part
        fFrameView,     // Tokenized View Type - icon, thumbnail etc.
        fSettingsPresentation, // Tokenized Presentation
        kODNULL);       // Source frame
```

RegisterWindowForFrame() is similar, but takes an existing root frame as a parameter.

```
window = windowState->RegisterWindowForFrame(ev,
            platformWindow,
            kODFrameObject,
            isRootWindow,
            isResizable,
            isFloating,
            shouldSave,
            shouldDispose,
            sourceFrame);
```

ODWindow Properties

Windows with the "is root" property set are windows that keep the document open. The shell closes the document when the last root window is closed. Most part editors will create a single root window in the Open() method, but some may create multiple root windows with different views of the document.

The "should save" property determines whether the window state saves the window persistently. This should generally be set to kODTrue for root windows, and kODFalse for non-root windows. Some parts may wish windows created using the View in Window command or the Open Selection command to be persistent. If this is the case, then additional work on the part of the part editor is required, because it must persistently maintain the connection between the source frame and the window, so that the window can be closed when the source frame is deleted.

The "frame type", "view type" and "presentation" properties are passed on to the root frame which the window creates. The presentation property can be used by the part editor to distinguish frames and hence windows from one another during event handling.

The "source Frame" parameter is used when an embedded frame is opened into a window. In other circumstances it will be kODNULL.

Window IDs

Part editors should not hang onto ODWindow pointers, because the shell or window state may close a window and invalidate the reference. Instead, the window state assigns IDs (valid for a session), and parts can make use of the following two methods:

```
ODID ODWindow::GetID();
ODWindow ODWindowState::AcquireWindow(in ODID windowID);

// Returns kODNULL if the window has been
// removed from the window state, i.e Closed.
```

Opening a Window

After creating and registering a window, a part editor will usually do the following:

```
  window->Open(ev);
  window->Show(ev);
  window->Select(ev);
```

The Open() method builds the facet hierarchy, but does not make the window visible.

The Show() method makes the window visible, but does not change the window ordering.

The Select() method brings the window to the front.

Closing Windows

Under normal circumstances, the OpenDoc Shell handles the closing of a document. If the window in question is the last root window of a draft, the draft is closed, along with all its windows. When a part editor registers a window, it indicates whether or not it is a root window.

If a window being closed is not a root window, or is not the last root window, the shell will just close that window.

Sometimes a part editor will need to close a window programmatically. For example, a part editor may wish to intercept clicks in the close box, and the Close menu item, and merely hide the window so that it can be more quickly displayed later. This may be appropriate for dialog windows, for example. In this case, the part editor will have to retain the window ID of the window, and close it programmatically at a later time. To close a window programmatically, a part editor should call ODWindow::CloseAndRemove(), after which the window object is no longer usable.

```
SOM_Scope ODID  SOMLINK ContainerPartOpen(ContainerPart *somSelf,
                                        Environment *ev, ODFrame* frame)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart","ContainerPartOpen");

    ODID windowID = 0;

    SOM_TRY

    TempODWindow window = kODNULL;
    if (frame) // Doing a View As Window or Open Root
    {
      if (frame->IsRoot(ev))  // Create Window For Root Frame
      {
         WindowProperties props;
         BeginGetWindowProperties(ev, frame, &props);

         ODULong flCreateFlags = props.createFlags ?
                                 props.createFlags :
                                 ODPlatformWindowDefaultCreateOptions;

         HWND hwndFrame = _fSession->CreatePlatformWindow(ev, flCreateFlags);
         // position the window to shape it was closed in
         // open from file in some position/properties as was saved
         WinSetWindowPos(hwndFrame, HWND_TOP, props.boundsRect.xLeft,
                                         props.boundsRect.yBottom,
                                         props.boundsRect.xRight,
                                         props.boundsRect.yTop,
                                         (SWP_SIZE |
                                         SWP_MOVE|
                                         props.swpFlags));

         window =  _fSession->
                   GetWindowState(ev)->
                   RegisterWindowForFrame(ev,
                                         hwndFrame,
                                         frame,
                                         props.isRootWindow, // Keeps draft open
                                         kODTrue,   // Is resizable
                                         kODFalse,  // Is floating
                                         kODTrue, // should save
                                         kODTrue, // should dispose
                                         props.sourceFrame);

         EndGetWindowProperties(ev, &props); // Release source frame
         window->Open(ev);
         window->Show(ev);
      }
      else  // View In Window
      {
         window = _fSession->GetWindowState(ev)->AcquireWindow(ev, _fWindowID);
         if (window)
            window->Select(ev);
         else
         {
            window = somSelf->CreateWindow(ev, frame);
            _fWindowID = window->GetID(ev);
            window->Open(ev);
            window->Show(ev);
            window->Select(ev);
         }
      }
    }
    else
    {
      window = somSelf->CreateWindow(ev, frame);
      _fWindowID = window->GetID(ev);
      window->Open(ev);
      window->Show(ev);
      window->Select(ev);
    }
    windowID = window->GetID(ev);

    SOM_CATCH_ALL
    SOM_ENDTRY

    return windowID;
```

```
}
SOM_Scope ODWindow*  SOMLINK ContainerPartCreateWindow(ContainerPart *somSelf,
                                                       Environment *ev,
                                                      ODFrame* sourceFrame)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart","ContainerPartCreateWindow");

    ODWindow* window = kODNULL;

    SOM_TRY

    Rect windRect;
    ODPlatformWindow platformWindow = kODNULL;

    // need to check if root frame
    if (sourceFrame && !sourceFrame->IsRoot(ev)) {
       SWP swp;
       swp.x = 100;
       swp.y = 100;
       swp.cx = 400;
       swp.cy = 400;

       platformWindow = _fSession->CreatePlatformWindow(ev,
                ODPlatformWindowDefaultCreateOptions |
                FCF_HORZSCROLL |
                FCF_VERTSCROLL);
       WinSetWindowPos(platformWindow, HWND_TOP,
                       swp.x, swp.y, swp.cx, swp.cy,
                       SWP_SIZE | SWP_MOVE);

     }
     else
     {
        platformWindow = _fSession->CreatePlatformWindow(ev,
                                                ODPlatformWindowDefaultCreateOptions);
    } /* endif */
    window =  _fSession->GetWindowState(ev)->
          RegisterWindow(ev, platformWindow,
                kODFrameObject,
                (sourceFrame==kODNULL),  // is root
                kODTrue,       // Is resizable
                kODFalse,       // Is floating
                kODTrue,       // should save
                kODTrue,       // should dispose
                _fPartWrapper,
                 _fSession->Tokenize(ev, kODViewAsFrame),
                 _fSession->Tokenize(ev, kODPresDefault),
                sourceFrame);
    SOM_CATCH_ALL
    SOM_ENDTRY

    return window;
}
```

Opening a Part into a Window

This document describes what you need to know to implement the Open() method of a part editor:

ODID Open(in ODFrame);

The Open() method is responsible for creating a window for the specified frame, and returning the ID of that window. If the window already exists, the method should simply activate it. The semantics are actually a little more complicated than this.

Open a part from nothing
When a new document is created from a template, it contains no window state. In this case the Open() method of the root part is called by the OpenDoc shell, passing kODNULL as the frame.

Open an embedded frame
When the user selects a frame and chooses Open Selection from the **Edit** menu, the active part calls the selected part's Open() method, passing in the selected frame.

Open a saved document
When a document (actually a draft) is saved, the draft contains a list of root frames of persistently stored windows. OpenDoc annotates each root frame with a storage unit containing the window properties (bounding rectangle, etc.). When the document is opened, and the Window State internalized, the Window State calls the Open method of each frame's part, passing in the frame. The implementor of the Open() method can distinguish this case because the frame is a root frame, and contains the annotations. Utility functions are provided to access the window properties.

Note: This means that windows are ALWAYS created by the parts, even when opening a saved document. Thus parts can and should do whatever they need to to adjust the window size and position to a given monitor configuration.

Open as
When the user activates a frame and chooses the **Open as** window, the active part calls its Open() method, passing in its frame.

See Closing Windows for a sample implementation of the Open() method from the Container Part.

Window Events

Events in the content region of a window are delivered to the appropriate part, which may be embedded several levels down. This document discusses events in the title bar and border of a window. See also the Basic Event Handling and Mouse Events recipes.

The OpenDoc Shell will handle events in the title bar and border of a window, including the system menu, minimize button and maximize button.

Close

When the user double-clicks on the system menu of a window, the OpenDoc shell closes the window, after which it can not be reopened. For a dialog window or palette, the part editor may wish to simply hide the window, so that it does not have to be recreated if the user shows it again.

Properties Notebook

Opendoc provides a standard Settings Notebook which is a subclass of ODExtension. The ODSettingsExtension class provides for a notebook control with a page(or pages) for displaying and modifying properties that are common across all parts, such as part name, part kind, etc. The default notebook includes an implementation for the following pages. A part kind page which allows the user to select the part kind and preferred part editor for the part. A **View As** page which allows users to select view type for a frame. A file attributes page which displays information about the part creation date/time as well as the size of the part in bytes. A file comments page which allows a user to view and edit part comments, key phrases and history. A general page which displays and allows editing of the the part's title and icon. If desired, a part can subclass the ODSettingsExtension and add pages for part specific properties.

Displaying the Properties Notebook

The properties notebook is displayed through the ODInfo object. The info object is instantiated when an Opendoc session object is created and can be accessed by calling the session's GetInfo method. The ODInfo object's ShowPartFrameInfo method then needs to be called with a reference to the appropriate facet of the part.

```
SOM_Scope ODBoolean  SOMLINK ContainerPartHandleMenuEvent(ContainerPart *somSelf,
                                                          Environment *ev,
                                                          ODFrame* focusFrame,
                                                          ODEventData* event)
```

```
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart","ContainerPartHandleMenuEvent");

    switch (command)
    {

     ..........(other commands go here)..........

       case VIEW_PROPERTIES:
       {
       ODFrameFacetIterator* facets = focusFrame->CreateFacetIterator(ev);
       facets->InitFrameFacetIterator(ev,focusFrame);
       _fSession->GetInfo(ev)->ShowPartFrameInfo(ev,facets->First(ev), kODFalse);
       delete facets;
       handled = kODTrue;
       }
       break;

    }
}
```

ODSettingsExtension Programming

The ODInfo object will query the part for an ODSettingsExtension. If a part does not provide one, the standard notebook will be displayed. For a part to provide its own settings extension it needs to provide implementation for the following ODPart methods.

```
SOM_Scope ODBoolean  SOMLINK ContainerPartHasExtension(ContainerPart *somSelf,
                                                       Environment *ev,
                                                      ODType extensionName)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart","ContainerPartHasExtension");

    if(!strcmp(extensionName, kODSettingsExtension))
       return kODTrue;
    else
       return ContainerPart_parent_ODPart_HasExtension(
              somSelf, ev, extensionName);
}
SOM_Scope ODExtension* SOMLINK ContainerPartAcquireExtension(
                              ContainerPart *somSelf,
                              Environment *ev,
                              ODType extensionName)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart","ContainerPartAcquireExtension");

    /*
     *   Note here that InitSettingsExtension should always be
     *   called with a reference to the part's wrapper, not a reference
     *   to the part itself.
     */
    if(!strcmp(extensionName, kODSettingsExtension))
    {
       if (!_fNotebook)
       {
          /* CNNotebook is a subclass of ODSettingsExtension */
          _fNotebook = new CNNotebook;
          _fNotebook->InitSettingsExtension(ev, _fPartWrapper);
       } /* endif */
       _fNotebook->Acquire(ev);
       return _fNotebook;
    }
    else
    {
       return ContainerPart_parent_ODPart_AcquireExtension(
              somSelf, ev, extensionName);
    } /* endif */
}

SOM_Scope void  SOMLINK ContainerPartReleaseExtension(ContainerPart *somSelf,
                                                      Environment *ev,
```

```
                                                    ODExtension* extension)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart","ContainerPartReleaseExtension");

    if (!extension) return;

    if(_fNotebook == extension) {
       _fNotebook->Release(ev);
       _fNotebook = 0;
    }
}
```

Subclassing ODSettingsExtension

A part can provide for its own settings extension notebook by subclassing ODSettingsExtension and overriding the AddNotebookPages method. When the notebook is to be displayed, the AddNotebookPages method is called, allowing the part to insert pages into the notebook. The derived class behavior for this method should be to call ODSettingsExtension::InsertNotebookPage for each additional page the part wants to add and then to call the base class implementation in order to to add the standard notebook pages. If a part wishes to omit or replace the standard settings pages, it can call the method for each standard page that it wishes to insert rather than calling the base class implementation of AddNotebookPages.

```
SOM_Scope ODULong SOMLINK CNNotebookAddNotebookPages(CNNotebook *somSelf,
                                                     Environment *ev,
                                                     HWND hwndNotebook,
                                                     ODULong flPageFlag,
                                                     ODULong ulReserve)
{
    CNNotebookData *somThis = CNNotebookGetData(somSelf);
    CNNotebookMethodDebug("CNNotebook","CNNotebookAddNotebookPages");

    SOM_TRY

    /* This method will add the background page behing the standard pages
    */

    somSelf->InsertBackgroundPage(ev, hwndNotebook);

    //we want the standard pages along with the icon/tree/details view pages
    CNNotebook_parent_ODSettingsExtension_AddNotebookPages(somSelf, ev,
                                        hwndNotebook,
                                        flPageFlag | kODFolderViewPages,
                                        ulReserve);

    SOM_CATCH_ALL
    SOM_ENDTRY

return flPageFlag;
}
```

Presentation Page

OpenDoc also provides an optional presentation page which allows a part to send a list of supported presentations to the notebook. A user can then choose between the options in a standard way. The procedure is to subclass the notebook as described above, and call the notebook's InsertPresentationPage method. A typelist with the presentation types supported is sent to this method as shown below, where a ClockPart allows a digital of analog presentation. The presentation page dialog procedure will call the part's PresentationChanged method with the user's choice.

```
SOM_Scope void  SOMLINK ClockSetAddNotebookPages(ClockSet *somSelf,
                                                 Environment *ev,
                                                HWND hwndNotebook)
{
```

```
     ODPart * part = (ODPart *)somSelf->GetBase(ev);
     ODTypeList * pres = part->GetStorageUnit(ev)->GetSession(ev)->

GetStorageSystem(ev)->CreateTypeList(ev,(ODTypeList*)kODNULL);
    pres->AddLast(ev, "Analog");
    pres->AddLast(ev, "Digital");

    somSelf->InsertPresentationPage(ev, hwndNotebook, prefs);
    ClockSet_parent_ODSettingsExtension_AddNotebookPages(somSelf, ev,
                                                         hwndNotebook);
    ODDeleteObject(pres);
}

SOM_Scope void  SOMLINK ClockPartPresentationChanged(ClockPart *somSelf,
                                                     Environment *ev,
                                                    ODFrame* frame)
{

    ClockPartData *somThis = ClockPartGetData(somSelf);
    ClockPartMethodDebug("ClockPart","ClockPartPresentationChanged");

    ClockPartInfoRec* pInfo = (ClockPartInfoRec*)frame->GetPartInfo(ev);

    ODSession* session = somSelf->GetStorageUnit(ev)->GetSession(ev);
    ODFrameFacetIterator* facets = frame->CreateFacetIterator(ev);
    ODFacet* facet = facets->First(ev);
    ODShape* usedShape = frame->AcquireUsedShape(ev, kODNULL);

    if(frame->GetPresentation(ev) == session->Tokenize(ev, "Digital"))
      pInfo->clock-> SetMode(DM_TIME | DM_DIGITAL | DM_SECONDHAND);
    else
    {
      /* analog is the default choice here */
      pInfo->clock-> SetMode(DM_TIME | DM_ANALOG | DM_SECONDHAND);
      frame->SetPresentation(ev, session->Tokenize(ev, kODPresDefault));
    }

    somSelf->Draw(ev, facet, usedShape);
    ODReleaseObject(ev, usedShape);
    delete facets;

    return;

}
```

Views

OpenDoc provides default implementation of icon/tree/details views. A part that supports these views only needs to capture the menu event and pass the view type requested by the user (icon, tree or details) on to an instance of the ViewExtension class via the call ODViewExtension::DisplayView.

If the view is DETAILS view, the part developer is able to add additional columns to be shown using the call ODViewExtension::AddDetailsColumns.

Displaying Icon/Tree/Details Views

```
SOM_Scope ODBoolean  SOMLINK
ContainerPartHandleMenuEvent(ContainerPart *somSelf, Environment *ev,
                             ODFrame* focusFrame, ODEventData* event)
{
    ContainerPartData *somThis = ContainerPartGetData(somSelf);
    ContainerPartMethodDebug("ContainerPart",
                             "ContainerPartHandleMenuEvent");
```

```
    ODBoolean handled = kODFalse;
    ODCommandID command = LONGFROMMP(event->mp1);

    switch (command)
    {
       ......(other command processing).....

       case VIEW_OATREE:
       case VIEW_OAICON:
       case VIEW_OADETAILS:
           {
              ODULong vType;
              if(_fViewExtension == kODNULL)
              {
                 _fViewExtension = new ODViewExtension;
                 _fViewExtension->InitViewExtension(ev,somSelf);
              }

              if (command == VIEW_OATREE)
                  vType = OD_TREEVIEW;
              else if (command == VIEW_OAICON)
                  vType = OD_ICONVIEW;
              else vType = OD_DETAILSVIEW;

              if(vType == OD_DETAILSVIEW)
                 Test_AddDetailsColumns(somSelf, ev, _fViewExtension);

              _fViewExtension->DisplayView(ev,somSelf,vType);
           }
                break;
```

When the view is DETAILS view, a part developer may add additional columns to display data from the embedded part's storage unit. Each of the columns in Details view represents a storage unit value. Below is an example of how to add a column only when all embedded parts of this container part are the same part kind.

Adding a New Column to the Details View

```
ODBoolean Test_AddDetailsColumns(ContainerPart *somSelf,
                           Environment *ev, ODViewExtension *ViewExt)
{
   ODEmbeddedFramesIterator *CurrentFrameIter = kODNULL;
   ODFrame *EmbeddedFrame = kODNULL;
   ODType kind;
   ODBoolean bSameKinds = FALSE;
   char *PropName, *ValName, *ColDataType;
   ODFIELDINFO NewField;

   PropName = new char[30];
   ValName = new char[30];
   ColDataType = new char[30];

   //This is an example of adding an additional column only when
   //all the embedded parts are of the same part kind - kSimplePartKind
   //If any embedded part is not of that part kind, no additional
   //columns are added; just the default columns are shown
   CurrentFrameIter=somSelf->CreateEmbeddedFramesIterator(ev,kODNULL);
   if (CurrentFrameIter) {
       EmbeddedFrame=CurrentFrameIter->First(ev);
       while (EmbeddedFrame) {
           kind = GetPartKind(EmbeddedFrame->AcquirePart(ev));
           if (strcmp(kind, kSimplePartKind)) {
              delete CurrentFrameIter;
              return FALSE;
           }
           EmbeddedFrame = kODNULL;
           bSameKinds = TRUE;
           if (CurrentFrameIter->IsNotComplete(ev))
                   EmbeddedFrame = CurrentFrameIter->Next(ev);
       }
       if (bSameKinds) {
          strcpy(PropName, kSimplePartNewProps);
```

```
        strcpy(ValName, kSimplePartNewPropValue);
        strcpy(ColDataType, kODISOStr);
        memset(&NewField, 0, sizeof(ODFIELDINFO));
        NewField.cb = sizeof(ODFIELDINFO);
        NewField.flDataAttrs = CFA_HORZSEPARATOR | CFA_LEFT |
                               CFA_SEPARATOR | CFA_STRING;
        NewField.flTitleAttrs = CFA_CENTER;
        strcpy(NewField.sTitleText, "New Column");
        ViewExt->AddDetailsColumns(ev, &NewField, &ColDataType,
                                   &PropName, &ValName, 1);
    }
}
```

Help

OpenDoc provides help panels for default base menu bar menu items. If a user presses F1 while keying through the menu items, an OD_HELP message is sent to the part. The HELP menu items are sent as WM_COMMANDS and should be handled as any other menu event.

If help was requested on a control field within a part or on a menu that was not created by OpenDoc, a WM_HELP message is passed to the part and should be handled the same way as it is in PM programming.

Displaying help for a menu item
This case statement is part of the HandleEvent method.

```
 case OD_HELP:
{
    ODHelp * help = _fSession->GetHelp(ev);
    USHORT  idSubTopic = SHORT2FROMMP(event->mp2);
    if((idSubTopic >= IDMA_COLOR_GRAY) &&
        (idSubTopic<=IDMA_COLOR_WHITE)){
       handled = kODTrue;
       switch (idSubTopic) {
         case IDMA_COLOR_GRAY:
            help->DisplayHelp(ev,"cnt.hlp",IDMA_COLOR_GRAY_PANEL);
            break;
         case IDMA_COLOR_RED:
            help->DisplayHelp(ev,"cnt.hlp",IDMA_COLOR_RED_PANEL);
            break;
         default:
            handled = kODFalse;
            break;
        } // endswitch
    } // endif
    break;
}
```

Undo

Undo under OpenDoc is multi-level and system-wide.

Note: The ODPart function calls, ReadActionState and WriteActionState are not supported by Undo in the current release of OpenDoc.

Parts

Parts should add undoable actions to the undo history using AddActionToHistory. whichPart identifies your part object, actionData is a reference to your data (a magic cookie), actionType is described below. undoActionLabel and redoActionLabel are user-visible strings corresponding to the undo and redo menu items.

```
void AddActionToHistory(in ODPart whichPart,
      in ODActionData actionData,
      in ODActionType actionType,
      in ODName undoActionLabel,
      in ODName redoActionLabel);
```

Here's some sample code:

```
// CREATE UNDO MENU ITEM TEXT
ODIText* undoActionName = CreateIText("Undo My Action");
ODIText* redoActionName = CreateIText("Redo My Action");

// CREATE A STRUCTURE TO HOLD DATA ABOUT ACTION
MyUndoInfo* uInfo = (MyUndoInfo*)AllocMem(sizeof(MyUndoInfo));
// ADD DATA TO STRUCTURE (TASK SPECIFIC)
uInfo->someData = ...

MyGetSession()->GetUndo()->AddActionToHistory(_fPartWrapper,
                                              (ODActionData)uInfo,
                                              kODSingleAction,
                                              undoMenuText,
                                              redoMenuText);

DeleteIText(undoActionName);
DeleteIText(redoActionName);
```

The above code illustrates adding a single action to the undo history. Undo makes a copy of the ODActionData and stores it away. To add a transaction to the undo history, such as the drag-moving of some data from one part to a another, parts must use the ODActionTypes, kODBeginAction and kODEndAction.

Note: Change in recipe for transactions:

The part initiating an action that may span multiple parts should place an action on the stack using the kODBeginAction constant for the ODActionType parameter to AddActionToHistory. When the action is complete, that same part should place the "end" action on the stack using the kODEndAction constant. In between these times, other parts may add single actions to the action history. These actions become part of the entire transaction.

For example, in the case of drag-moving of some data from one part to a another, the source part should add a begin action to the action history, the receiving part should add item(s) to the undo stack about the data received using the kODSingleAction constant, and finally, the source part can close out the transaction by adding an end action to the history.

Another example: in the case of a paste with a link, the destination part should add the begin and end actions to the stack and the source part should add one or more single actions to the undo history.

If the part placing the begin and end actions on the stack notices an error at any point after placing the begin action onto the stack, it should roll back the transaction. Currently there is no clean way to do this other than to clear the entire undo history. We will mostly likely add a new functionality in the near future to allow for rolling back a transaction.

When an action is undone, ODPart::UndoAction is called for the part that furnished the action. Similarly ODPart::RedoAction is called when an action is redone. When a transaction is undone or redone, every action between the beginning and ending actions, as well as those actions themselves, will be undone or redone.

ODPart::DisposeActionState is called whenever an item on the undo or redo stack is permanently removed. You will be passed a copy of the ODActionData that you originally added with AddActionToHistory. You must take whatever action is relevant to your part. This may be the disposal of any auxiliary data associated with this undo action.

Using Resources in OpenDoc

Resources(Dialogs/Menus/Bitmaps/Icons/Accelerators/Messages) are linked into your parts DLL by using IBM's CSet/2 Resource Compiler (RC).

Sample code used to read a resource object from your resource data.

```
DosQueryModuleHandle(THISDLL,hMod);              // Get DLL Handle
DosGetResource(hmod,ulTypeID,ulNameID,ppObject)  // Get Resource Address

  ... Use of the resource Object

DosFreeResource(ppObject);                       // Free Resource
```

Sample Code used to Load a Dialog from your resource data.

```
DosQueryModuleHandle(THISDLL,hMod);              // Get DLL Handle
hwndDlg = WinLoadDialog(HWND_DESKTOP,HwndOwner,dlgProc,
                        hmod,DLG_ID,NULL);
WinProcessDlg(hwndDlg);                          // Process Dialog
```

Sample Code used to Load a Menu from your resource data.

```
DosQueryModuleHandle(THISDLL,hMod);              // Get DLL Handle
Menu = WinLoadMenu(HWND_OBJECT,hmod,MenuID);

fMenubar->AddMenuLast(ev,MenuID,Menu,myPart);
```

Sample Code used to Load Accelerators from your resource data.

```
DosQueryModuleHandle(THISDLL,hMod);              // Get DLL Handle
hAccel = WinLoadAccelTable(Hab,hmod,IDofAccelTable);
WinCopyAccelTable(hAccel,&PartAccel,ulCopyMax);
fMenuBar->AddToAccelTable(ev,num_accels,&PartAccel);
```

Facet Windows

When an ODPlatformWindowCanvas object is created for an on-screen, dynamic canvas (using ODFacet::CreatePlatformWindowCanvas), OpenDoc creates and initializes a Presentation Manager window for the root facet which is a child window of the Document Shell client window. The class of this window (in the PM sense) is registered by OpenDoc. The window class name for facet windows is OpenDoc:FacetWindow, and this class is registered as a private class on a per process basis. A separate window of this class is created for each facet that is contained within the root facet.

It is important to note the distinction here between the concept of a window in the OpenDoc sense, and the use of Presentation Manager windows in OpenDoc. The OpenDoc class ODWindow and the type ODPlatformWindow refer to desktop level document windows. The ODPlatformWindow type on OS/2 is the window handle of the document frame window. The OS/2 implementation of OpenDoc uses Presentation Manager windows for facets that are displayed in an on-screen, dynamic canvas. These windows are specific to the OS/2 implementation of OpenDoc and have no relationship to the ODWindow class.

The parent-child relationship of facet windows corresponds to the embedding hierarchy of the facets. For example, the facet window of a facet that is embedded within its containing facet is a child window of the containing facet's facet window. There is no such correspondence, however, for sibling and size-position relationships. The Z-order of facet windows that are children of the same facet window does not necessarily correspond to the Z-order of the associated facets. When an embedded facet is created, the associated facet window is positioned beneath all other PM windows that are also children of the facet window for the containing facet. Your part editor does not need to be concerned with the Z-order of the facet windows of embedded facets except when your part's facet window also contains non-OpenDoc PM windows such as buttons and scroll-bars (more on this later).

The size and position of facet windows has no relationship to the size and position of the associated facets except for the root facet. Every facet window in a document is positioned at location (0,0) relative to its parent window and every facet window has the same size as its parent window. When the root facet is resized (by resizing the document window) OpenDoc resizes the facet window of the root facet, and every embedded facet, accordingly. The position of a facet window, however, always remains anchored to the bottom-left corner of its parent window, and hence, the bottom left corner of the document window. Your part editor should never modify the size or position of facet windows, however, you do need to be aware of this behavior in order to properly position other windows that your part creates such as buttons and scroll bars.

This decoupling of the facet window size-position from the facet size-position may seem odd at first, but it allows the same drawing recipe to

be used for all types of canvases. It is accomplished by having parts (instead of PM) be responsible for doing all of the required clipping when drawing. This is part of the basic drawing recipe for parts on all OpenDoc platforms.

Querying the Facet Window Handle

All facets that are descended from a root facet with an on-screen, dynamic canvas have a unique facet window. This is true even if the facet in question has (or is descended from a facet that has) an off-screen canvas, however, in this case the facet windows for facets that are displayed in the off-screen canvas will not be visible. This is done by making the facet window of the facet with the off-screen canvas a child of the desktop object window. If the off-screen canvas is later removed, then the facet windows for the facets that were displayed in that canvas are again made visible by making the facet window of the facet that had the off-screen canvas a child of its containing facet's facet window.

You can query the window handle (*HWND*) of the facet window for a given facet by calling ODFacet::GetFacetHWND. If the facet has a facet window, the *HWND* of the window will be returned, otherwise NULLHANDLE is returned. For the reasons discussed above, you should not use the *HWND* of a facet window for drawing your part's content. Instead, you should use the presentation space obtained from your facet's canvas. See the Part Drawing recipe for more information.

Given a facet window handle, you can obtain a ponter to the ODFacet object that is associated with the window by calling the class method M_ODFacet::clsGetFacetFromHWND. This call will succeed only if the facet object is in the same process.

Specifying the PS for a Facet Window

By default, calling ODPlatformCanvas::GetPS for a platform canvas object that has a window returns a cached presentation space for the specified facet's facet window. Because this is a cached PS, it must be released using ODPlatformCanvas::ReleasePS when you are done drawing. You should not keep a reference to this PS beyond the scope of the method where it was obtained. This means that you must set up any attributes, transforms and clipping that are required every time you want to draw you part's content. The Focuslib class which is provided in the public utilities will help out with this, but you may want to set a micro or normal presentation space and set these values only when they change. You can do this by calling ODPlatformCanvas::SetPS in your part's FacetAdded method. You can then set the clip region and transform when you are notified that they have changed in your part's GeometryChanged method. Calling ODPlatformCanvas::ReleasePS for a non-cached PS has no effect.

The following example shows how you can set a presentation space for a facet window in your part's FacetAdded method.

```
SOM_Scope void SOMLINK MyPartFacetAdded(MyPart* somself,
                                                    Environment* ev,
                                                    ODFacet *facet)
{
  SOM_TRY

  HWND hwnd;
  HDC hdc;
  HPS hps;
  ODPlatformCanvas *platformCanvas;
  platformCanvas = facet->GetCanvas(ev)->GetPlatformCanvas(ev, kODGPI);
  if (platformCanvas->HasWindow(ev))
  {
     // Get the window handle and DC
     hwnd = facet->GetFacetHWND(ev);
     hdc = WinOpenWindowDC(hwnd);

     // create micro PS and associate DC
     SIZEL sizl = {0, 0};
     hps= GpiCreatePS(WinQueryAnchorBlock(hwnd), hdc, &sizl,
                      PU_PELS | GPIT_MICRO | GPIA_ASSOC);

     // associate PS with the facet
     platformCanvas->SetPS(ev, hps, facet);
  }

  SOM_CATCH_ALL
  SOM_ENDTRY
}
```

Whenever the canvas for one of your facets is changed, your part will be called in its CanvasChanged method. It is possible that your facet was created with an off-screen canvas, and that canvas was removed so now your facet uses the root facet's window canvas. In this case, you may need to follow the steps above to set the presentation space for your facet from within your CanvasChanged method. You can

determine if this is necessary by calling ODPlatformCanvas::HasWindow and if this method returns kODTrue, then call ODPlatformCanvas::GetPS to obtain the presentation space for the facet window. You can then query the type of the presentation space using GpiQueryPS and if the type is not GPIT_MICRO because you have previously set it using ODPlatformCanvas::SetPS, you may do so now. Don't forget to call ODPlatformCanvas::ReleasePS to release the cached PS that was obtained with the call to GetPS first, or the call to SetPS will fail.

The following example shows how you can set the clip region and default viewing transform for a non-cached PS in your part's GeometryChanged method. You could also set default attributes based on the facet's highlight mode in your part's HighLightChanged method. Refer the file FOCUSLIB.CPP in the public utilities to see what is done to prepare the presentation space for drawing when a CFocus object is instantiated.

```
SOM_Scope void SOMLINK GeometryChanged(MyPart* somself,
                                       Environment* ev,
                                       ODFacet *facet,
                                  ODBoolean clipShapeChanged,
                                  ODBoolean externalTransformChanged)
{
  SOM_TRY

  ODPlatformCanvas platformCanvas;
  platformCanvas = facet->GetCanvas(ev)->GetPlatformCanvas(ev, kODGPI);
  if (platformCanvas->HasWindow(ev))
  {
     HPS hps = platformCanvas->GetPS(ev, facet);
     if (clipShapeChanged)
     {
        HRGN hrgnClip, hrgnOld;
        TempODShape tempShape =
           facet->AcquireAggregateClipShape(ev, kODNULL);
        TempODShape clipShape = tempShape->Copy(ev);
        TempODTransform xform =
           facet->AcquireContentTransform(ev, kODNULL);
        clipShape->Transform(ev, xform);
        hrgnClip = clipShape->CopyRegion(ev);
        GpiSetClipRegion(hps, hrgnClip, &hrgnOld);
        GpiDestroyRegion(hps, hrgnOld);
     }
     if (externalTransformChanged)
     {
        MATRIXLF mtx;
        TempODTransform xform =
           facet->AcquireContentTransform(ev, kODNULL);
        xform->GetMATRIXLF(ev, &mtx);
        GpiSetDefaultViewMatrix(ev, 9, &mtx, TRANSFORM_REPLACE);
     }
  }

  SOM_CATCH_ALL
  SOM_ENDTRY
}
```

Now, when you're called in your Draw method, the presentation space for your facet window is already set up and it's not necessary to set the clip region, viewing transform and other attributes. You need to check the canvas type, though, to determine if it's a window canvas. In this example, we use the C language functions BeginFocus and EndFocus rather than the C++ class CFocus so that we can conditionally invoke EndFocus. Because EndFocus is not automatically invoked when it goes out of scope as in the CFocus destructor, we should make sure that it gets called if an exception is thrown. See the Exception Handling recipe for more information on handling exceptions in OpenDoc.

```
SOM_Scope void  SOMLINK MyPartDraw(ContainerPart *somSelf,
                           Environment *ev,
                           ODFacet* facet,
                           ODShape* invalidShape)
{

   MyPartData *somThis = MyPartGetData(somSelf);
   MyPartMethodDebug("MyPart","MyPartDraw");
    CFocus* foc = 0;

   SOM_TRY
    HPS hps;
    ODPlatformCanvas* platCanv =
             facet->GetCanvas(ev)->GetPlatformCanvas(ev, kODGPI);

    if (platCanv->HasWindow(ev))
       hps = platCanv->GetPS(ev, facet);
    else
       foc = new CFocus(ev, facet, invalidShape, &hps);
```

```
    // Draw your part's content here

    // Clean up
    if (foc)
       ODDeleteObject(foc);

   SOM_CATCH_ALL
    if (foc)
       ODDeleteObject(foc);
   SOM_ENDTRY
}
```

Using Embedded PM Controls within a Facet

You can embed PM controls within a facet that has a facet window. For the purpose of this discussion, the term "PM control" is used to represent any Presentation Manager window that sends notification messages to its owner. You embed a PM control by creating the control using the WinCreateWindow function and specifying the facet window for your part's facet as the parent and owner of the control window. You should not specify the WS_CLIPSIBLINGS style flag for the window. The following sample code creates a button window.

```
HWND hwndFacet, hwndButton;
if (hwndFacet = facet->GetFacetHWND(ev))
{
   hwndButton = WinCreateWindow(hwndFacet, WC_BUTTON, "Push",
                                BS_PUSHBUTTON | WS_VISIBLE
                                0, 0, 10, 30,
                                hwndFacet, HWND_TOP, 0, 0, 0);

   facet->SetPartInfo(ev, (ODInfoType)hwndButton);  // Save handle
}
```

Notification messages sent to the facet window will be forwarded to the facet's part in its HandleEvent method. You can identify these messages as coming from the facet window by the *hwnd* field in the event data structure. For normal events (those sent to your part by the OpenDoc dispatcher) the *hwnd* field will contain the handle for the document shell client window. For messages sent by a facet window, this field will contain the window handle of the facet window.

You position the control windows within your facet based on the facet's window frame transform, or the window content transform, depending upon whether you want the control anchored to a position in your part's content or on your display frame.

Managing Clipping

It is your part's responsibility to manage the clipping of embedded PM controls. You do this by calling the WinSetClipRegion function, specifying the window handle of the PM control and the region to be used for clipping, in the control's window coordinates. You calculate the clip region for the PM control based on your part's aggregate clip shape and, optionally, any embedded frames or content elements that obscure it.

The following sample code positions the button created above at the point (100,100) in content coordinates and sets the window clip region. The ideal place to call this code would be within your part's GeometryChanged method.

```
HWND hwndButton = (HWND)facet->GetPartInfo(ev);
if (hwndButton)
{
   TempODTransform transform =
                     facet->AcquireWindowContentTransform(ev, kODNULL);

   // We want to display the button if our frame transform specifies
   // only translation and scale (no reflection, rotation or shear) becuase
   // these operations cannot be performed on PM windows.

   if (transform->GetType(ev) < kODLinearXform )
   {
      ODPlatformCanvas pcanv;
```

```
        // Set position of button window

      ODPoint origin(ODIntToFixed(100), ODIntToFixed(100));
      transform->TransformPoint(ev, &origin);
      POINTL ptl = origin.AsPOINTL();
      WinSetWindowPos(hwndButton, 0,
                      origin.x, origin.y, 0, 0,
                      SWP_MOVE);

        // Set clip shape of button window

      TempODShape clipShape = ODCopyAndRelease(ev,
                     facet->AcquireWindowAggregateClipShape(ev, kODNULL);
      clipShape->Transform(ev, transform);

      HRGN hrgnClip = clipShape->CopyRegion(ev);
      pcanv = facet->GetCanvas(ev)->GetPlatformCanvas(ev, kODGPI);
      HPS hps = pcanv->GetPS(ev);
      origin.x = -origin.x;

        // convert region to window coords of button window

      origin.y = -origin.y;
      GpiOffsetRegion(hps, hrgnClip, &origin);
      pcanv->ReleasePS(ev);
      WinSetClipRegion(hwndButton, hrgnClip);
      WinShowWindow(hwndButton, TRUE);
   }
   else
   {
      // We can't properly display the button with the given transform.
      // Hide the window and we'll draw the text into the frame in our Draw
      // method

      WinShowWindow(hwndButton, FALSE);
   }
}
```

In general, the Z-order of the PM controls and the facet windows that are children of your part's facet window does not matter to OpenDoc because sibling windows are not clipped by PM (none of the windows have the WS_CLIPSIBLINGS style). Rather, your part determines the apparent Z-order of embedded frames, PM controls and content elements by controlling the clipping and/or order of updating of these items. Facet windows created by OpenDoc for your part's embedded facets are children of the facet window of your part's facet and placed at the bottom of all other child windows. OpenDoc does not attempt to further manage the Z-order of facet windows.

Although OpenDoc does not care about the Z-order of the windows that are children of your part's facet window and the window Z-order is generally not significant with regard to window updating, there is one important exception. PM controls may not be able to properly display target emphasis during a drag and drop operation if they are not above all the sibling facet windows, as well as any other window to which they should not be clipped. This is because the PS returned by the DrgGetPS function that is used to draw target emphasis during drag and drop is always clipped by sibling windows. OpenDoc parts use the new DrgGetClipPS function which allows the caller to specify clipping flags and works very similar to WinGetClipPS. To allow PM controls that don't use this new function to properly display target emphasis during drag and drop, your part should maintain the window Z-order of the windows that are children of you part's facet window such that all the PM controls are above all the facet windows, and the Z-order of the PM controls, relative to each other, is the same as the apparent Z-order that you achieve by clipping or controlling drawing order.

Your part controls what elements of your content (embedded parts, PM controls and content objects) appear to be above other elements in one of two ways. The first way is to control the clipping of these elements. You do this by clipping the drawing of each element by those objects that are to obscure it. For embedded facets, you specify a clip shape for the facet by calling the facet's ChangeGeometry method. For PM controls, you specify a clip region for the window by calling WinSetClipRegion. For content objects, you clip the drawing of your content during your Draw method. When you use clipping to manage Z-order, you do not need to be concerned with the updating of your embedded parts or embedded PM controls. You simply draw your own content (clipped by your facet's aggregate clip shape) in your part's Draw method, and OpenDoc will take care of making sure that embedded parts and PM controls are updated as needed.

The second method of controlling the apparent Z-order of content elements is to control the order in which these elements are updated. The elements are not clipped to each other, but are simply drawn in order from bottom to top. This technique is known as the "painter's algorithm." You cause an embedded facet to be updated by calling that facet's Update method. You cause an embedded PM control to be updated by calling the WinUpdateWindow function, specifying the window handle of the PM control. You update content elements by simply drawing them. If you use the painters algorithm, you do not need to clip your content and embedded PM controls by obscuring content elements, but you still need to clip them by your facet's aggregate clip shape. An alternative to setting the window clip region of each of your embedded PM controls is to set the window clip region of the facet window of your part's facet. You do this using the facet's aggregate clip shape, transformed by the facet's content transform. Since the clip region set by WinSetClipRegion clips the specified window and all of that window's child windows by the specified region, this has the effect of clipping your part's embedded PM controls as well as your part's content by the facet's aggregate clip shape. By convention, only the facet's part should set the window clip region of its facet window.

The painter's algorithm will produce the appearance of flickering during updating as areas of the screen that are overlapped by content elements are painted multiple times. This technique can be more efficient than clipping, though, especially if clipping would require complex clip shapes. One major drawback to the painter's algorithm is that it only works if the embedded parts and PM controls do not update their

display asynchronously. A PM button window, for example, updates its display when the mouse is clicked on it. If the button is not clipped by overlapping content, it may paint over other content elements that are supposed to be above it in the Z-order.

Using embedded PM controls can make your part more functional, however, you should always be prepared to render your part without the controls if they cannot be displayed properly. There are several situations in which you would not want to (or would not be able to) display embedded controls. One of these situations is if your part's facet has a frame transform that specified shear or rotation. Since there is no way to shear or rotate a PM window, you should display some alternate representation of your part's content that does not use the PM controls. Another situation where embedded PM controls cannot be displayed is if your part's facet is displayed in an off-screen canvas. In this case, your facet will still have a facet window and you can create the embedded PM controls but they will not be displayed. You should draw a representation of your part's content into the off-screen canvas that does not make use of the embedded controls.

Handling Mouse Events

By default, mouse events that occur over embedded PM controls are handled by the controls. Your part can specify that it wants to handle mouse events that occur over embedded controls by calling the SetHandleMouseEvents method of the containing facet. You can specify that you want to handle mouse events (excluding drag/drop events), drag/drop events or both. The effect of specifying you want to handle these events in a facet is to make all the embedded controls within that facet transparent to these events.

Implementing a Dispatch Module

The OpenDoc Dispatcher can be extended by developers to handle additional event types, perhaps associated with some exotic input device like a glove.

ODDispatcher maintains a dictionary of dispatch modules, with the event type as the key. When ODDispatcher::Dispatch() is called, the dispatcher looks up a dispatch module for the supplied event type, and calls its Dispatch() method.

To extend the dispatcher, a developer must implement a subclass of ODDispatchModule, and install it in the dispatcher.

Defining a Dispatch Module

ODDispatchModule is a System Object Model (SOM) class, just like ODPart, and must be subclassed (in IDL). The subclass should have an Init<ClassName> method, and should override the Dispatch() method:

```
ODBoolean Dispatch(inout ODEventData, inout ODEventInfo)
```

The Init<ClassName> method should call InitDispatchModule, and the Dispatch() method will presumably locate a part to handle the event, and call its HandleEvent() method.

There is a method Redispatch() defined in ODDispatcher. When the standard OpenDoc dispatch module transforms an event (for example WM_COMMAND into OD_PRINT), it redispatches. This means that dispatch module patches can intercept the transformed event. The Dispatch() method of ODDispatchModule has an eventInfo parameter containing the additional information associated with some of the transformed events (such as an embedded frame).

A dispatch module should be installed by calling ODDispatcher::AddDispatchModule:

```
void AddDispatchModule(in ODEventType eventType,
                       in ODDispatchModule dispatchModule);
```

For example:

```
myDispatchModule = new MyDispatchModule();
myDispatchModule->InitMyDispatchModule(ev, session);
dispatcher->AddDispatchModule(ev, kFooBarEvent, myDispatchModule);
```

The installation could take place in a part's initialization method, or in a Shell Plug-In.

Monitors

A dispatch module can also be installed as a monitor, by calling ODDispatcher::AddMonitor() instead:

```
void AddMonitor(in ODEventType eventType, in ODDispatchModule dispatchModule);
```

Unlike regular dispatch modules, multiple monitors can be installed for a single event type. The dispatcher calls the Dispatch() method of all monitors for an event before calling the Dispatch() method of the single regular dispatch module for that event type. The Boolean result returned by the monitors' implementation of Dispatch() is ignored. In other words, a monitor cannot interfere with the regular dispatching of the event, unless it actually changes the event type on the fly [Shudder].

Implementing a Focus Module

The OpenDoc Arbitrator can be extended by developers, to handle additional focus types, perhaps associated with some exotic peripheral device.

A focus is simply an ISO standard string. CI Labs will presumably set up a process for registering new foci for public consumption. The OpenDoc functions use tokenized forms of these strings (using ODSession::Tokenize()).

ODArbitrator maintains a dictionary of focus modules, with the tokenized focus as the key. The methods of ODArbitrator (like RequestFocusSet()) look up a focus module for each focus, and calls appropriate methods of the focus module.

To extend the Arbitrator, a developer must implement a subclass of ODFocusModule, and install it in the arbitrator.

Defining a Focus Module

ODFocusModule is a System Object Model (SOM) class, just like ODPart, and must be subclassed (in IDL). The subclass should have an Init<ClassName> method, and should override the methods below. The Init<ClassName> method should call InitFocusModule.

```
ODBoolean IsFocusExclusive(in ODTypeToken focus);
void SetFocusOwnership(in ODTypeToken focus, in ODFrame frame);
void UnsetFocusOwnership(in ODTypeToken focus, in ODFrame frame);
void TransferFocusOwnership(in ODTypeToken focus,
        in ODFrame transferringFrame,
        in ODFrame newOwner);
ODFrame AcquireFocusOwner(in ODTypeToken focus);
ODFocusOwnerIterator CreateOwnerIterator(in ODTypeToken focus);
ODBoolean BeginRelinquishFocus(in ODTypeToken focus,
          in ODFrame requestingFrame);
void CommitRelinquishFocus(in ODTypeToken focus,
        in ODFrame requestingFrame);
void AbortRelinquishFocus(in ODTypeToken focus,
        in ODFrame requestingFrame);
```

The class reference documentation describes the semantics of each method in detail, and the source code for OpenDoc's standard exclusive focus module is included with these recipes. See ExFocus.idl and ExFocus.cpp.

Installing a Focus Module

A focus module should be installed by calling ODArbitrator::RegisterFocus:

```
void RegisterFocus(in ODTypeToken focus,
            in ODFocusModule focusModule);
```

For example:

```
// this is assuming that somSelf is your ODPart object
ODSession *Session = somSelf->GetStorageUnit(ev)->GetSession(ev);
ODArbitrator *Arbitrator = Session->GetArbitrator(ev);

 const ODFocusType kFooFocus = "Foo";
 ODTypeToken FooFocus = Session->Tokenize(ev,kFooFocus);
 ODMyFocusModule *myFocusModule = new ODMyFocusModule;
 MyFocusModule->InitFocusModule(ev,Session);

 Arbitrator->RegisterFocus(ev,FooFocus,myFocusModule);
```

The installation could take place in a part's initialization method, or in a Shell Plug-In.

Part Persistency

The part persistency recipes are listed as follows:

- Multiple Kind Support
- Part Storage Model
- Part Init and Externalizing
- Part Info 'Ternalization
- Display Frame 'Ternalization
- Lazy Frame Internalization

Multiple Kind Support

OpenDoc provides a structured storage model which enables part editors to persistently store multiple representations of their content. In order to present this feature in a simple and easy to use manner to users, part editors must follow a few rules in general, and be more particular in their implementation of a handful of their methods. This document contains the list of part editor responsibilities, and sample code which would allow a part editor to fulfill these responsibilities.

Terminologies

The different terminologies that you need to be familiar with are described as follows:

Kind	Specific data format or type
Category	Set of similar kinds. A kind can be in multiple categories.
Preferred kind of a part	The preferred kind of a part designates the data format which the editor running the part must use to 'ternalize the part's content. The preferred kind is stored in kODPropPreferredKind property of the part's storage unit. If the kODPropPreferredKind property does not exist, then the editor can presume that the preferred kind of the part is the value type of the first value in the contents property (kODPropContents) of the part's storage unit.

Registration	Registration of a part handler. See the Dynamic Binding recipes for more detail.

Standard Kinds

Standard kinds are those data formats which, either through an official decree or by some de facto means, have become widely used and accepted. Here are some examples:

Industry Standard Kinds	ASCII, TIFF, JPEG, etc.
Standard OS/2 Kinds	kODKindPlainText, kODKindImageGIF, etc.

Supporting a standard kind has many advantages.

It permits your part editor to support a large amount of the content already out there. It permits users to read old documents with your part editor. Users like being able to read their old documents with the latest software.

Supporting a standard kind means there is a greater chance that your editor can read parts created by other editors. This enables users to share documents created by your part editor.

Note on supporting standard OS/2 kinds:
Part handlers can register to support these standard kinds, as well as file types and file extensions.

Coding Required

In order to correctly support multiple kinds, there are a handful of methods in the part function in which your editor needs to make sure it behaves properly.

InitPart

This method is called when a template is created. Your editor should make sure it does the following:

1. Set the preferred kind. Write the appropriate ISO string into the kODPropPreferredKind property of your part.
2. Create a value whose type is that kind in the contents property
3. Initialize the content value with valid initial contents. Often leaving the value empty is a valid; it depends on the particular data format. Alternatively, perhaps as an optimization, your editor may just set an internal dirty flag, and then perform the above three steps in its externalize method when it realizes there is no contents property.

InitPartFromStorage

This method is called when a part is read back in. This happens whenever a user opens a document or creates a new document. Your editor should make sure it does the following:

1. If you support any PlatformKinds (file), then you should first check for the kODFileTypeEA and kODFileType value types in the contents property. If they are there, then you have been bound to an empty storage unit which is pointing to a file which you should use to internalize from. This can happen if the user has dragged and dropped a traditional file onto an OpenDoc document and if your part editor was bound to the drop. For detailed information on how to make this work, see the Drag and Drop recipe specifically the Incorporating Data from a Non-OpenDoc Document section.

2. Get the preferred kind. Read the value from the kODPropPreferredKind property of your part. If no property exists, then use the value type of the first value of the contents property as the preferred kind. Keep the preferred kind in a field.

3. Focus your part's storage unit to the value of the contents property whose value type is the preferred kind.

4. Read the contents of that first value and create the in-memory data structures necessary to represent that content.

   Note: It is possible that your editor was bound to a part which previously had a different editor. In this case, the binding subsystem will automatically notify the user. In this case, if your editor does not support the preferred kind, then use the highest fidelity kind in the contents property which your editor does support as the de facto preferred kind. Do not update the preferred kind property until externalize or ChangeKind is called on your part.

Externalize

This method may be called at any time. Depending on the Save model of the current document, and the idle-time optimizations which may or may not be present, your part may be told to externalize only when the user Saves a document, or as often as every minute. The point is, do not assume that when you are told to externalize, that it is for any one particular reason. As an optimization, your editor should probably keep an fDirty flag which is set whenever the user changes the part's content, and cleared whenever Externalization is completed. If your fDirty flag is clear then your externalize method should be a no op. Your editor should make sure it does the following:

1. Clean up the storage unit. You need to prepare the storage unit for clean externalization from your part editor. You should only have to do this the first time externalize is called on your part whenever it is brought into memory. Remove any values that are not being updated. Specifically, this means remove any values which have value types/kinds which your editor does not support, or are of lower fidelity than kinds which you wish externalize.

2. Add values if necessary. Use AddValue to create or recreate the value types you wish to externalize in proper fidelity order (first value - highest fidelity to last value - lowest fidelity). Fidelity ordering is important, because OpenDoc binding and translation use the ordering of the values in the contents property to determine which value is probably of the highest fidelity to determine which editor would best edit any given part. These first two steps are called *prepping your storage unit* ; there will be examples in other methods where you need to reprep your storage unit.

3. Externalize your content in the format of the preferred kind which your editor kept track of in a local field. Do not write some other kind instead; that would be implicit translation, or translating formats behind the user's back. It is usually not a good idea to change things behind the user's back. Some applications behave this way today. The application claims to read/write a particular data format, but when a document of that kind is opened with that application, the application converts the document to its own proprietary format when you save. This is really annoying to the user, who is left wondering why cannot their documents be more stable, and stick with the format they were created with. In today's applications, this unexpected format change is also often associated with resetting of the document's name to "Untitled x" or "FooDocument - converted". In OpenDoc, parts do not have any control over the name of the document, so this errant behavior is automatically prevented. The name of the document, just like the preferred kind of a part, should be considered a user setting. Editors should not tamper with user settings. There are ways for a user to explicitly change the preferred kind of a part. These will be discussed later. For now keep in mind that only the user should be changing the preferred kind of a part, and only then through an explicit action.

4. OPTIONAL: Write out alternative kinds. As discussed above, your editor may want to write out one or more kinds in addition to the preferred kind. The typical part editor should by default only write out either the one preferred kind, or the preferred kind, and one interchange (or standard) kind. If you want your editor to write out many more representations than that, we recommend that the editor present a user interface in their settings or preferences dialog to allow the user to pick a set of kinds which your editor externalizes by default.

ExternalizeKinds

This method is called when the user does a **Save a copy** as multiple formats, or when a data interchange utility goes through a document asking all the parts to externalize themselves in a set of kinds extracted from a set of standard interchange kinds. Your editor should make sure it does the following:

1. Externalize the set of kinds specified. Make sure that the fidelity ordering of the values in your contents property is maintained by creating the values for these kinds in the right order. You may need to reprep your contents property and recreate the values in order to make sure they are in the appropriate fidelity ordering. Be sure to write out these kinds in addition to the preferred kind, not instead of the preferred kind.

2. Ignore unsupported kinds. Ignore any kinds in the set which you do not support.

ChangeKind

This method is called when the user changes the preferred kind of a part. This is usually done from the Part/Document Properties notebook, although do not assume it is the only user interface that can cause this method to be called. Your editor should make sure it does the following:

1. Externalize the part in the given kind. Make sure that the fidelity ordering of the values in your contents property is maintained by creating the values for these kinds in the right order. You may need to reprep your contents property and recreate the values in order to make sure they are in the appropriate fidelity ordering. It is up to your part editor whether you keep the previous preferred kind or not.

2. Set that kind to be the preferred kind. Write the given kind into the preferred kind property of the part.

CloneInto

This method is similar to externalize. It is usually called in response to a data interchange action such as using the clipboard or drag and drop. See the Data Interchange recipes for precise detail about implementing the CloneInto method. For the purposes of the multiple kind support however, make sure your editor does the following:

1. Write the same kinds you if you were Externalizing plus any standard kinds you support. It is more important to write out standard kinds during CloneInto than externalize because when your CloneInto method is called, it is more likely that your part is in the middle of a data interchange operation, and should therefore be trying extra hard to enable the user to move content to a different editor or application.

2. SetPromiseValue for each kind if you are using promises.

Resources Required

Your part editor tells OpenDoc what kinds it supports in response to its clsGetODPartKinds metaclass method. This method should return a sequence of PartKindInfo structures that contain the appropriate information for each part kind. See the Part Handler Registration recipe for details on how to register a part handler.

Multiple Kinds User Interface Description

Earlier we stated that the user should be the only thing which changes the preferred kind of a part. Here is how they do it. To change the preferred kind of an embedded part, the user does the following:

1. Select the part.
2. Choose **Selection Properties** from the Edit menu.
3. Select from the choices in the "Kind:" listbox of the Type page.
4. Optionally change the editor; some editors work better than others when it comes to certain kinds.

To change the preferred kind of the root part document, the user does the following:

1. Choose **Document properties** from the Document menu.
2. Select from the choices in the "Kind:" listbox of the Type page.
3. Optionally change the editor; some editors work better than others when it comes to certain kinds.

In any discussion of preferred kinds, it makes sense to also describe preferred editors. Using the OpenDoc Preferences notebook the user

can set a particular editor to be the preferred editor for a particular kind. Whenever the user encounters a part of that kind, whose previous editor is unavailable, the user set preferred editor for that kind is used to run that part. Similarly, the user can set a particular editor to be the preferred editor for a whole category, such as Other Plain Text. This means that whenever the user encounters a part whose kind is in the Plain Text category, whose previous editor is unavailable, and there is no preferred editor for that kind, the user set preferred editor for that category (Plain Text) is used.

Part Storage Model

Parts, like all other persistent objects have a storage unit in which they can store their content data and state persistently. However, parts are intimately involved in actions like binding, translation, data interchange (clipboard, drag and drop). As a result, parts need to satisfy many more requirements than the other persistent objects, and so they need to be more disciplined about how they use their storage units.

Rules and Requirements

1. Parts need to be able to store multiple representations of their content, ordered by fidelity.
2. Each representation of a part's content must be a full representation, that is it cannot just be a delta of another representation.
3. It must be possible to extract exactly one representation from the part's storage unit, without understanding the format of any of the representations in the part.
4. It must be possible to remove all but one representation from the part's storage unit, without understanding the format of any of the representations in the part.
5. It must be possible to distinguish annotations/meta-information on/about the part, and content of the part.
6. Annotations/meta-information on/about a part may go away at any time, independent of the part editor, and therefore cannot be depended on to store critical content. However, they can be used to store non-critical optimizations such as caches.
7. Property names are ISO strings which constrain the naming conventions we can place on properties to that of ISO strings. The only naming conventions we can apply to ISO strings is that of prefix ownership, that is TC corporation may decide and control the format and meaning of all ISO strings beginning with TC.

Details of the Part Storage Model

As a result of the above rules/requirements, parts store their contents as follows:

- Every part creates a contents property, the constant for which (kODPropContents) is defined in StdProps.IDL.
- For every representation it wishes to store, the part creates a value, the value type of which is the kind of the representation.

  These values are ordered by fidelity, for example highest fidelity first, lowest fidelity last.

  Into each of these values, the part streams out its contents using the appropriate data format.

  One representation, one value.

No content should be stored outside of the contents property.

All other properties on a part's storage unit are for meta-information/annotations and should not contain content.

Example:

```
property OpenDoc:Property:Contents
         value OSA:Kind:TestDrag
         value OSA:Kind:StyledText
```

```
value OSA:Kind:PlainText
```

Q and A

- Q: But how do you tell the difference between annotations and content?

  A: There is often a misconception that there is a lot of overlap between annotations and content. In fact, it is difficult to come up with a precise definition which separates the two. However, more often than not, it is fairly easy to categorize any particular piece of information as annotation or content.

  Here are some questions to ask yourself about a piece of info X which you are unsure of whether it is annotation or content.

  - Would users be upset if they thought they had saved X and it disappeared?
  - Do you want X to be persistent across machines?
  - Do you want X to be persistent across editors?
  - Do you want X to be persistent across platforms (for cross-platform formats)?

  If you answered yes to any of the above, then X is content.

- Q: But cannot we just distinguish meta-information from content by using a property naming convention, and store them both as sets of top level properties?

  A: No. Storing content outside the contents property would violate requirements 3 and 4 above. Also, placing a more restrictive naming convention on property names would violate requirement 7 above.

- Q: But what if a part wants to store a particular representation using more than one property in a structured way?

  A: Any part may store a particular representation in a structured fashion by requesting additional storage units from the draft and tying them together with references.

- Q: Would not it be more efficient to keep the multiple properties in the part's storage unit instead of requesting another storage unit?

  A: It is hard to be more efficient when it does not even work! That approach would violate requirements 3 and 4. It is not a question of efficiency, it is a question of functionality.

- Q: Usually when an application stores a document, it stores some data in the data fork of the document file and then a bunch of other information as resources in the resource fork. Isn't the part storage model more restrictive than this?

  A: No. The part storage model is only as restrictive as you want it to be. If, like most applications, your part uses only a single stream to store its contents, then your storage model fits right into the part storage model, you simply use a single value to store your contents. However, if you are using resources or some other form of structured storage, then you can ask your storage unit's draft object for additional storage units, and then store references to these storage units inside your primary content value. See the persistent reference document for more information about using storage unit references. It is important to note that because you can always ask for more storage units and then save references to them in any value, you can construct any kind of arbitrary graph structure for your storage representation. This is much more powerful than current day files and streams. Each additional storage unit can have multiple properties (you may think of these as multiple forks), and each property can have multiple values (again, different values within the same property should only be used for multiple representations of the same data).

  Also, note that:

  1. Most applications use only the data fork.
  2. On DOS, Windows, UNIX and any other major OS out there, the files have a single fork, and therefore their applications use only a single flat stream for storing document content. Cross platform applications which want document portability also use only the data fork on OS/2. This single flat stream to store one representation is the model we are leveraging. This the 90% (if not more) case of applications which are out there today.

Part Init and Externalizing

This document contains some sample code showing how a part editor externalizes and internalizes itself.

The sample code shown here is for a very simple part editor. Its assumptions may not be valid for a part editor with more functionality:

- PartData is a Pascal string (Str255) which holds the part content or part Data.
- The sample code reads and writes the entire part Data from and to persistent storage.
- The sample code uses a fDirty flag which it sets to true when its contents change.
- The sample code uses a fSelf field to hang onto the part-wrapper object.

Proper Treatment of Permissions

Note that the part is responsible for caching the draft permissions in InitPartFromStorage. The sample code does this by assigning a field:

```
fPermissions = storageUnit->GetDraft(ev)->GetPermissions(ev);
```

in the InitPartFromStorage. Whenever the potential for changing user content exists, the part should check to make sure fPermissions >= kDPSharedWrite before changing any of the content. Also, as a side effect, if fPermissions < kDPSharedWrite then fDirty should never be able to become kODTrue, and as a result, the externalize method will never write to the contents property.

PartWrapper

In addition to a storage unit, InitPart and InitPartFromStorage methods on ODPart also takes an extra parameter, *partWrapper* . The *partWrapper* parameter represents a delegator object which OpenDoc uses to insulate the rest of OpenDoc from having to have a direct pointer to your ODPart object. This feature helps implement features such as being able to change the editor of part without closing and reopening the document. Plus, it makes an excellent bottleneck for platform developers. For the most part, all the part wrapper does is delegate methods of the part function to your ODPart object.

The part editor must save this parameter in the part's internal field; you will see this in the sample code below. Whenever an OpenDoc function requires the part editor to pass in an ODPart* representing the part, instead of passing in somSelf, the part editor must pass in *partWrapper* stored in the part's internal field. An example is when registering the part for idle time. A part editor should never pass somSelf in as a parameter to an OpenDoc function call.

Sample Code for Initialization

```
// Includes the appropriate .XH files from OpenDoc
#include "StorageU.xh"
#include "StdProps.xh"
#include "POUtils.h"
  .
  .
  .
// Define the part kind.
// This should probably be in your part's ....def.H file which should
// be #included  in your .r file for your nmap resources
#define kMyPartKind "Sample:Kind:CMyKind"
  .
  .
  .
void MyPartInitPart(MyPart* somSelf,
                    Environment* ev,
                    ODStorageUnit* storageUnit,
                    ODPart* partWrapper)
{
  .
  .
```

```
  .
  // Call your parent's init routine
  parent_InitPart(somSelf, ev, storageUnit, partWrapper);

  // Set your dirty flag
  fDirty = kODTrue;

  // Save off the part-wrapper object
  fSelf = partWrapper;
}
void MyPartInitPartFromStorage(MyPart* somSelf,
                               Environment* ev,
                               ODStorageUnit* storageUnit,
                               ODPartWrapper* partWrapper)
{
  // Initialize the parent
  parent_InitPartFromStorage(somSelf, ev, storageUnit, partWrapper);

  // Focus to the value where the part Data is stored
  storageUnit->Focus(ev,
                     kODPropContents,
                     kODPosUndefined,
                     kMyPartKind,
                     0,
                     kODPosUndefined);

  // Get the size of the string
  fPartData[0] = storageUnit->GetSize(ev);

  // Get the string itself
  storageUnit->GetValue(fPartData[0],&(fPartData[1]));

  // Save off the part-wrapper object
  fSelf = partWrapper;

  // Note the draft permissions, in order to avoid
  // writing to a read-only draft
  fPermissions = storageUnit->GetDraft(ev)->GetPermissions(ev);

  // Set the dirty flag to false
  fDirty = kODFalse;
}
```

Sample Code for Externalization

```
void MyPartExternalize(MyPart* somSelf,
Environment* ev)
{
  // Only externalize when the part has been updated.
  // Externalize the parent.
  parent_Externalize(somSelf, ev);

  if (fDirty)
  {
    // Get the part's storage unit
    ODStorageUnit* storageUnit = somSelf->GetStorageUnit(ev);
    if (storageUnit->Exists(ev, kODPropContents, kMyPartKind, 0))
    {
      // Focus to the desired value
      storageUnit->Focus(ev,
                         kODPropContents,
                         kODPosUndefined,
                         kMyPartKind,
                         0,
                         kODPosUndefined);
    }
    else
    {
      storageUnit->AddProperty(ev, kODPropContents);
      storageUnit->AddValue(ev, kMyPartKind);
```

```
      // Adding a property and value has the side effect of focusing
      // the storage unit to that value.
    }
    // Write out the part data
    storageUnit->SetValue(ev, fPartData[0],&(fPartData[1]));

    // Set the dirty flag to kODFalse to signify that the part is clean
    fDirty = kODFalse;
  }
}
```

Part Info 'Ternalization

Parts can attach part info data to their display frames. This data can be used to hold any information the part wants to associate with the display frame, and is stored persistently in the frame's storage unit. While the data is stored with the frame, the part editor is responsible for its 'ternalization, as the editor is the only entity that understands the structure and format of the part info.

Persistent Format

A frame's part info is stored in the frame's storage unit in the part info property. The standard name for this property is found in the constant kODPropPartInfo. Just as a part editor stores multiple representations of a part's contents in multiple values in the contents property, the editor should also store multiple representations of the part info which correspond to the representations of the contents. Some content formats may not have corresponding part info, especially interchange formats like JPEG or RTF. However, many editors will define formats with corresponding part info, and if your editor supports another editor's native format directly, it should also support the part info for that format.

The names of part info formats which are linked to a content format should be the same as the name of the content format, with the segment PartInfo appended to the end of the ISO name. For example, if your part content format was SurfSoft:SurfText 1.0 the part info should be SurfSoft:SurfText1.0:PartInfo.

For a part which externalized representations for the formats SurfText 1.0, RTF, ClarisWorks 3.0, and ASCII, its display frames should have part info with formats SurfSoft:SurfText1.0:PartInfo and Claris:ClarisWorks 3.0:PartInfo (assuming the interchange formats have no associated part info).

'Ternalizing Part Info

There are three calls in the part function that deal with 'ternalization of part info:

- ReadPartInfo: internalizes the part info from persistent storage
- WritePartInfo: externalizes the part info into persistent storage
- ClonePartInfo: externalizes the part info as in WritePartInfo, but clones object references

In all of these calls, the frame passes a StorageUnitView which is focused to the frame's part info property, but not to any value within that property. Your part editor must get the view's storage unit, and focus it to value(s) in that property as necessary to read or write the formats it needs to.

When to Write

When externalizing a property of a persistent object, it is only necessary to write the data if it is the first time writing it, or if it has changed from the last time the object was externalized. As an optimization, you can keep an isDirty flag in a frame's part info, indicating whether it has changed from when it was internalized or last externalized. In WritePartInfo, if the flag is false, skip externalizing; if the flag is true, externalize the data and set the flag to false. Note that you must always externalize part info when cloning (in ClonePartInfo), as you are making a new

copy.

Just as a part must mark the draft as dirty when its content needs to be externalized, it must also mark the draft dirty when part info needs to be externalized. The part should call draft->SetChangedFromPrev() when part info has changed and needs to be externalized. Note that changes to part info in a non-persistent frame should not mark the draft dirty.

Sample Code

This code 'ternalizes a simple part info structure. The only value in it is an RGB color describing the background color of the frame. The forDebuggingRGBColor intermediate value makes debugging easier.

```
#include "ISOStr.h"
#include "StorageU.xh"
#include "SUView.xh"
#include "StorUtil.h"
  .
  .
  .
ODInfoType CMyPart::ReadPartInfo(Environment* ev,
                                 ODFrame* frame,
                                 ODStorageUnitView* storageUnitView)
{
  ODStorageUnit* su = storageUnitView->GetStorageUnit(ev);
  ODPropertyName propName = storageUnitView->GetProperty(ev);
  PartInfoRec* partInfo = kODNULL;

  if (ODSUExistsThenFocus(ev, su, propName, kKindSamplePartInfo))
  {
    partInfo = new PartInfoRec;

    RGBColor forDebuggingRGBColor ;
    StorageUnitGetValue(su,
                        ev,
                        sizeof(RGBColor),
                        (ODValue)&forDebuggingRGBColor);

    partInfo->bgColor = forDebuggingRGBColor;
    partInfo->isDirty = kODFalse;
  }
  SOMFree(propName);
  return (ODInfoType)partInfo;
}
void CMyPart::WritePartInfo(Environment *ev,
                            ODInfoType partInfo,
                            ODStorageUnitView* storageUnitView)
{
  if (partInfo && partInfo->isDirty)
  {
    ODStorageUnit* su = storageUnitView->GetStorageUnit(ev);
    ODPropertyName propName = storageUnitView->GetProperty(ev);
    ODSUForceFocus(ev, su, propName, kKindTestDraw);

    RGBColor forDebuggingRGBColor ;
    forDebuggingRGBColor = ((PartInfoRec*)partInfo)->bgColor;
    StorageUnitSetValue(su,
                        ev,
                        sizeof(RGBColor),
                        (ODValue)&forDebuggingRGBColor);

    partInfo->isDirty = kODFalse;
    SOMFree(propName);
  }
}
void CMyPart::ClonePartInfo(Environment *ev,
ODInfoType partInfo,
ODStorageUnitView* storageUnitView)
{
  if (partInfo)
  {
    ODStorageUnit* su = storageUnitView->GetStorageUnit(ev);
    ODPropertyName propName = storageUnitView->GetProperty(ev);
    ODSUForceFocus(ev, su, propName, kKindTestDraw);

    RGBColor forDebuggingRGBColor ;
    forDebuggingRGBColor = ((PartInfoRec*)partInfo)->bgColor;
```

```
    StorageUnitSetValue(su,
                        ev,
                        sizeof(RGBColor),
                        (ODValue)&forDebuggingRGBColor);

    SOMFree(propName);
  }
}
```

Display Frame 'Ternalization

A well behaved part editor needs to keep track of its display frames, and store and retrieve them from an annotation property on its storage unit. This document contains the list of methods affected, and a brief description of their responsibilities. The sample code documents step by step how to implement this functionality in those methods.

Caveats

The following sample code:

- Contains minimal proper error checking.
- Does not account for the case where a part may want to do AbstractTertiaryCase.
- Uses the CLink and CLinkedList classes, which are implemented by the part.
- Uses StorUtil.H and StdTypIO.H which are unsupported utilities of OpenDoc. It is recommended that you copy the sample code from these unsupported utilities into your project.
- Does not contain any optimizations such as lazily internalizing your display frames, or only rewriting your list of display frames when it changes.
- May not contain all the necessary #includes.
- Is written with the assumption that the rest of the implementation follows the SOM Wrapper/C++ implementation strategy.
- Does not contain full implementation for the methods listed.

Methods Affected

Constructor	Clear fDisplayFrames.
Destructor	Delete fDisplayFrames if it is not null, and set it to null.
InitPart	Create a new, empty collection (fDisplayFrames) in which to store a list of your display frames.
DisplayFrameAdded	Add the frame passed in to fDisplayFrames.
DisplayFrameConnected	Add the frame passed in to fDisplayFrames if it is not already in the list.
DisplayFrameRemoved	Remove the frame passed in from fDisplayFrames.
DisplayFrameClosed	Remove the frame passed in from fDisplayFrames.
Externalize	If necessary create a DisplayFrames annotation property and StorageUnitRefs value on your storage unit. Otherwise just focus to your DisplayFrames annotation property and its StorageUnitRefs value. Write out a list of references to your display frames from fDisplayFrames.
InitPartFromStorage	Focus to your DisplayFrames annotation property and its StorageUnitRefs value. Read in the list of your display frames into fDisplayFrames. See the Lazy Frame Internalization recipe for detail.

Sample Code

```
void CMyPart::CMyPart()
{
  .
  .
  .
  fDisplayFrames = kODNULL;
  .
  .
  .

}
void CMyPart::~CMyPart()
{
  .
  .
  .
  if (fDisplayFrames != kODNULL)
  {
    delete fDisplayFrames;
    fDisplayFrames = kODNULL;
  }
  .
  .
  .
}
void CMyPart::InitPart(Environment* ev,
                       ODStorageUnit* storageUnit,
                       ODPart* partWrapper)
{
  .
  .
  .
  // Create a list to keep track of the frames we are being
  // displayed in
  fDisplayFrames = new CLinkedList;
  .
  .
  .
}
void CMyPart::InitPartFromStorage(Environment* ev,
                                  ODStorageUnit* storageUnit,
                                  ODPart* partWrapper)
{
  .
  .
  .
  // Create a list to keep track of the frames we are being
  // displayed in
  fDisplayFrames = new CLinkedList;

  // Read in our list of display frames
  storageUnit->Focus(ev,
                     kODPropDisplayFrames,
                     kODPosUndefined,
                     kODWeakStorageUnitRefs,
                     0,
                     kODPosUndefined);
  ODULong size = storageUnit->GetSize(ev);
  storageUnit->SetOffset(ev, 0);

  for (ODULong offset = 0; offset < size; offset += sizeof(ODStorageUnitRef))
  {
    StorageUnitGetValue(storageUnit,
                        ev,
                        sizeof(ODStorageUnitRef),
                        weakRef);
```

```
    if (storageUnit->IsValidStorageUnitRef(ev, weakRef))
    {
      ODFrame *frame = storageUnit->
                       GetDraft(ev)->
                       AcquireFrame(ev,   // Calls my DisplayFrameConnected
                       storageUnit->
                       GetIDFromStorageUnitRef(ev, weakRef));
      frame->Release(ev);
    }
  }
  .
  .
  .
}
void CMyPart::DisplayFrameAdded(Environment* ev,
                                ODFrame* frame)
{
  .
  .
  .
  frame->IncrementRefCount(ev);
  fDisplayFrames->Add((ODPtr)frame);
  .
  .
  .
}
void CMyPart::DisplayFrameConnected(Environment* ev,
                                    ODFrame* frame)
{
  .
  .
  .
  if (!fDisplayFrames->Contains((ODPtr)frame))
  {
    frame->IncrementRefCount(ev);
    fDisplayFrames->Add((ODPtr)frame);
  }
  .
  .
  .
}
void CMyPart::DisplayFrameRemoved(Environment* ev,
                                  ODFrame* frame)
{
  .
  .
  .
  fDisplayFrames->Remove((ODPtr)frame);
  frame->Release(ev);
  .
  .
  .
}
void CMyPart::DisplayFrameClosed(Environment* ev,
                                 ODFrame* frame)
{
  .
  .
  .
  fDisplayFrames->Remove((ODPtr)frame);
  frame->Release(ev);
  .
  .
  .
}
void CMyPart::Externalize(Environment* ev)
{
  .
  .
  .
  ODStorageUnitRef weakRef;
  ODStorageUnit* su = this->GetStorageUnit(ev);
  ODSUForceFocus(su, kODPropDisplayFrames, kODWeakStorageUnitRefs);

  // Persistent object references are stored in a side table, rather than
  // in the property/value stream.
  // Thus, deleting the contents of a value will not delete the references
  // previously written to that value.
  // To completely delete all references written to the value, we must
  // remove the value and add it back.
  su->Remove(ev);
  su->AddValue(ev, kODWeakStorageUnitRefs);
```

```
  for (CLink* link = fDisplayFrames->First();
       link->Content();
       link = link->Next())
  {
    frame = (ODFrame*) link->Content();
    frameID = frame->GetID(ev);

    // Write out weak references to each of the part's display frames
    su->GetWeakStorageUnitRef(ev, frameID, weakRef);
    StorageUnitSetValue(su, ev, sizeof(ODStorageUnitRef), weakRef);
  }
  .
  .
  .
}
void CMyPart::ReleaseAll(Environment* ev)
{
  .
  .
  .
  ASSERT(fDisplayFrames->First() == kODNULL);
  .
  .
  .
}
```

Lazy Frame Internalization

Winning through laziness:
There are situations in OpenDoc where eagerly doing work as soon as possible is not the best thing to do, and the work should be deferred until it really needs to be done. Internalization is usually a good candidate for being lazy. In general, do not internalize data until it is needed, or you can predict it will be needed soon (for example you might pre-fetch the next page so scrolling will be fast). A particular case of lazy internalization involves your part's frames, both display frames and embedded frames.

Since OpenDoc places very few requirements on a part's internal representation, your part is free to organize its data however it pleases. The only requirements relevant to frames are that your part must be able to supply an EmbeddedFramesIterator for its embedded frames, and it should externalize a list of its display frames in a standard property of its storage unit.

For very simple containing parts, there will not be much advantage in lazily internalizing embedded frames. But for large containing parts, the advantages can be quite significant. For example, a containing part which embeds five thousand frames (think CD-ROM here) would not want to internalize all of those frames at the same time. There would be no way to display all the frames simultaneously, and a large amount of memory would effectively be wasted on keeping those persistent objects resident. All parts should consider lazy internalization of display frames, since your part has no control over the number of display frames that may be added to it.

How To Be Lazy

The main trick to lazy frame internalization is stopping before you go too far. In InitPartFromStorage, you should stop short of calling draft->GetFrame(). Instead, just hang on to the frame ID that you got from the storageUnitRef. (Remember that display frame references should be weak, so they may not actually have a frame or ID at the other end). The minimal way to deal with this ID is to create a structure that can hold a frame pointer and a frame ID. Anywhere in your code you would have stored a frame reference, store one of these structures instead. If the structure does not have a frame yet, use the ID to get it, then store it in the structure for next time (Remember to increment the frame's reference count.)

For slightly more effort up-front, you can have a much easier time of it later. You can create a class that stores a frame reference and a frame ID, and does the GetFrame for you when necessary. The class Futon below is an example of such. (Do not try to read any significance into the name. It is just a random word that was easy to type).

Getting Laziness to Work for You

Now that your part is more relaxed about internalizing its frames, here is how it can take advantage of being lazy.

- Don't get a frame unless you really need to. You have already fixed your part so that instead of direct frame references in its content, it has Futons instead. Now it would be a real waste if you went and made all those Futons get their frames right away. As a containing part, your part probably has a pretty good idea of where its embedded frames are located, and how big they are.- You need this information to determine when to add a facet to a frame to make it visible. So be lazy and do not get the embedded frame from the Futon until you need to add a facet to it, or perform some other operation on it. Just do not get it for the sake of getting it.

- Purge frames when memory is low. When your part gets a Purge() call, it should attempt to free up memory. If you are set up for lazy internalization, it is easy to purge frames. This only works if you do not keep direct frame references anywhere. First off, only purge frames that are not currently visible. Start by removing a frame's facets. Then call Release() on the frame, and null it out in the Futon. If a frame's embedded and containing parts both do this, the reference count of the frame should drop to zero, and the object can be flushed from memory. Don't call Close() on an embedded frame to purge it from memory. It is an awful lot of work to Close and reconnect frames, and the technique described above works just as well if not better.

Sample Code

```
#define  INCL_ODAPI
#define  INCL_ODFRAME
#define  INCL_ODDRAFT
#include <os2.h>
#include "ODUtils.h"
class Futon
{
public:
  Futon(ODDraft* draft, ODID frameID);
  ~Futon();

  ODFrame* GetFrame(Environment* ev);
  void Purge(Environment* ev);

  ODDraft* fDraft;
  ODID fFrameID;
  ODFrame* fFrame;
};
Futon::Futon(ODDraft* draft, ODID frameID)
{
  // Note:
  // I am not reference counting the draft, because the draft is
  // referenced from the part's storage unit, so it is
  // not going away while the Futon (and thus the part)
  // is still around.
  fDraft = draft;
  fFrameID = frameID;
  fFrame = kODNULL;
}
Futon::~Futon()
{
  ODSafeReleaseObject(fFrame);
}
ODFrame*
Futon::GetFrame(Environment* ev)
{
  if (fFrame == kODNULL)
    fFrame = fDraft->GetFrame(ev, fFrameID);
    return fFrame;
}
void
Futon::Purge(Environment* ev)
{
  ODReleaseObject(ev, fFrame);
}
```

Dynamic Binding

The dynamic binding recipes are listed as follows:

Binding

The OpenDoc name binding subsystem is responsible for determining what part editor or part viewer, of those installed on user's system, is to be loaded in order to edit or view a given part or document. The binding process uses information gathered from nmap resources found in any installed editors or viewers.

Terminologies

Kind	String (ISO type format) which is used to identify a data format. Examples of kinds: SurfCorp:SurfText SurfCorp:Picture:BlackAndWhite SurfCorp:Picture:Color
Category	Kinds can be grouped together in one or more categories. Examples of categories: OpenDoc:Category:Text OpenDoc:Category:Graphics OpenDoc:Category:Table Kinds may be in more than one category.
Part editor	Dynamically linked library which has the ability to manipulate (read, Sample, edit, print, write) a number of different kinds of parts. Since the context implies it, editor is often used instead of part editor. An editor is said to exist, or to be, if it has been installed.
Part viewer	Part editor which is not capable of changing the contents of parts.
Binding	Act of taking a part and assigning a part editor to it at run time.
Unbound	Without a binding (to an editor).
Fidelity	Different kinds of data can be said to represent the same information, but some may be qualitatively better or worse than others. A qualitatively better kind is said be of higher Fidelity. There is no universal ordering of kinds by fidelity, therefore fidelity is always determined by a particular context. Part editors support many different kinds, and stored with the part editor is a list of those kinds in fidelity ordering. Parts may contain many different kinds, different representations of their content, and they are fidelity ordered by what the part editor which last wrote out that part considers fidelity.

Installation of OpenDoc Software

This section describes the location on your computer systems you should install the OpenDoc software that you create followed by a description for registering a part handler.

OpenDoc creates two folders on your Desktop. These are the OpenDoc Template and OpenDoc Shell Plug-In folders. The registration and template functions create a template object in the OpenDoc Template folder. OpenDoc Extensions which are to be used as Shell Plug-In should be copied to the OpenDoc Shell Plug-In folder.

Install part editors and viewers to either the OS2\DLL directory or a new one of your choice. If a new directory is created then it should be added to the libpath in the CONFIG.SYS.

Part Handler Registration

This recipe describes where on users' computer systems you should install the OpenDoc software that you create and a description for registering a part handler.

OpenDoc creates two folders on the user's Desktop. These are the OpenDoc Templates and OpenDoc Shell Plug-Ins folders. The registration and template functions create a template object in the OpenDoc Templates folder. OpenDoc Extensions which are to be used as Shell Plug-Ins should be copied to the OpenDoc Shell Plug-Ins folder.

Install part editors and viewers to either the OS2\DLL directory or a new one of your choice. If a new directory is created then it should added to the libpath in the CONFIG.SYS.

Registering a Part Handler

All part handlers are now required to register themselves.

- Each part handler library (DLL) must contain the information that is required to register all of the parts that it contains.
- The SOM interface and implementation repositories must be updated during registration. A SOM interface repository must be supplied by the part developer at registration time so that OpenDoc's SOM interface repository can be updated.
- The registration data base will be created and maintained by one or more DSOM objects running in one process on each machine.

The changes to the part handler's IDL are simple additions that can be plugged in from the sample code given below. The following three steps need to be done:

1. Add a metaclass definition
2. Add a interface prototype for the metaclass
3. Set the part handler's metaclass

Start by adding the new metaclass for the part. In order to register a part handler with OpenDoc, the part handler must respond to several method calls during the registration process. These method calls are defined in an ODPart metaclass called M_ODPart. The M_ODPart's methods are used in the part handler so that the part handler can respond to the registration processes requests for information particular to the part handler.

Add the following code after the part handler's implementation definition:

```
// Added for Part Registration
interface M_YourPart : M_ODPart
{
  #ifdef __SOMIDL__
  implementation
  {
    functionprefix = M_YourPart;
    override:
    clsGetODPartHandlerName,
    clsGetODPartHandlerDisplayName,
    clsGetODPartKinds,
    clsGetOLE2ClassId,
    clsGetWindowsIconFileName;
  };
  #endif
};
```

From the sample code above, the text YourPart should be replaced with your part handler's class name. This code sets the metaclass to inherit from M_ODPart and overrides the specified five methods.

The next change is to add the metaclass's interface prototype. Place the following code after all INTERFACE prototype lines in the IDL. If there are no INTERFACE prototypes, place this code before the part handler's interface definition:

```
interface M_YourPart;
```

The final addition is in the part handler's implementation definition. Add this line between the FUNCTIONPREFIX and MAJORVERSION lines of the implementation definition:

```
metaclass = M_YourPart;
```

Now, save the .IDL file, backup the .CPP file, delete the .XH and .XIH files and recompile.

After updating the .CPP file with the latest SOM headers for the overridden metaclass methods (SOM compiler does this), the following code can be added to the .CPP file. There are five methods that need to be altered and placed at the bottom of the .CPP file. In addition, there are five new CONST definitions used with the following code that need to be placed at the top of the .CPP file after all the #INCLUDE statements:

```
const ODType   kPartHandlerName             = "YourPart";
const ODType   kPartHandlerDisplayName      = "One's Part";
const ODType   kKindTestYourPart            = "YourPart:yourdll";
const ODType   kYourPartKindDisplayName     = "One's Part Kind";
const ODType   kYourPartCategory            = "Test Part";
const ODType   kYourPartCategoryDisplayName = "Category Name";
    .
    .
    .
SOM_Scope ISOString SOMLINK
M_YourPartclsGetODPartHandlerName(M_YourPart *somSelf,
                                  Environment *ev)
{
  /* M_YourPartData *somThis = M_YourPartGetData(somSelf); */
  M_YourPartMethodDebug("M_YourPart",
                        "M_YourPartclsGetODPartHandlerName");
  char * handlerName = kPartHandlerName;
  return (ISOString) handlerName;
}
SOM_Scope string SOMLINK
M_YourPartclsGetODPartHandlerDisplayName(M_YourPart *somSelf,
                                         Environment *ev)
{
  /* M_YourPartData *somThis = M_YourPartGetData(somSelf); */
  M_YourPartMethodDebug("M_YourPart",
                        "M_YourPartclsGetODPartHandlerDisplayName");
  string displayName = kPartHandlerDisplayName;
  return displayName;
}
SOM_Scope _IDL_SEQUENCE_PartKindInfo SOMLINK
M_YourPartclsGetODPartKinds(M_YourPart *somSelf,
                            Environment *ev)
{
  /* M_YourPartData *somThis = M_YourPartGetData(somSelf); */
  M_YourPartMethodDebug("M_YourPart",
                        "M_YourPartclsGetODPartKinds");
  _IDL_SEQUENCE_PartKindInfo YourPartInfo;

  // Create structure PartKindInfo  and allocate memory for variable
  PartKindInfo * info = (PartKindInfo *)SOMMalloc(sizeof(PartKindInfo));
  info->partKindName  = (ISOString) SOMMalloc(strlen(kKindTestYour)+1);
  info->partKindDisplayName = (string)SOMMalloc(strlen
                                    (kYourPartKindDisplayName)+1);
  info->filenameFilters =  (string)SOMMalloc(strlen("")+1);
  info->categories    = (string)SOMMalloc(strlen(kYourPartCategory)+1);
  info->categoryDisplayName = (string)SOMMalloc(strlen
                                    (kYourPartCategoryDisplayName)+1);

  // New part template support
  info->filenameTypes = (string)SOMMalloc(strlen("")+1);
  info->objectID = (string)SOMMalloc(strlen("")+1);

  // Copy the information into the structure
  strcpy(info->partKindName , kKindTestYour);
  strcpy(info->partKindDisplayName, kYourPartKindDisplayName);
  strcpy(info->filenameFilters, "");
  strcpy(info->categories, kYourPartCategory);
  strcpy(info->categoryDisplayName, kYourPartCategoryDisplayName);

  // New part template support
  strcpy(info->filenameTypes, "");
  strcpy(info->objectID, "");
  YourPartInfo._maximum = 1;
```

```
  YourPartInfo._length = 1;
  YourPartInfo._buffer = info;
  return YourPartInfo;
}
SOM_Scope string  SOMLINK
M_YourPartclsGetOLE2ClassId(M_YourPart *somSelf,
                            Environment *ev)
{
  /* M_YourPartData *somThis = M_YourPartGetData(somSelf); */
  M_YourPartMethodDebug("M_YourPart",
                        "M_YourPartclsGetOLE2ClassId");
  string classID = "";
  return classID;
}
SOM_Scope string  SOMLINK
M_YourPartclsGetWindowsIconFileName(M_YourPart *somSelf,
                                    Environment *ev)
{
  /* M_YourPartData *somThis = M_YourPartGetData(somSelf); */
  M_YourPartMethodDebug("M_YourPart",
                        "M_YourPartclsGetWindowsIconFileName");
  string fileName = "";
  return fileName;
}
```

Throughout this sample code, the YourPart should be replaced with the name of the part handler. All of the pertinent information that is going into the registry is stored in the CONST definitions defined earlier, so once those definitions have been configured to the part handler, the rest of the code changes are simply name changes, with the exception of the *filenameFilters* data in the PartInfo structure, which can change to match any file types associated the part handler. For example, TunePart, a part which edits tunes, creates and handles files with the .TUN extension, so *.TUN was placed in-between the quotes in the clsGetODPartKinds method.

The final step is to create the registration script. Below is an example REXX version. Simply change this code to the part handler's information.

```
====================================================================
BEGIN REXX INSTALLATION CMD FILE
====================================================================
/*******************************************************************/
/*                                                                 */
/*  FileName: YourInst.CMD                                         */
/*                                                                 */
/*  Purpose:  Installation command file for One's Part.            */
/*                  Calls the OpenDoc REXX function                */
/*                  to register the part.                          */
/*                                                                 */
/*******************************************************************/

/* Add the registration function (and utilities) */
SAY " Adding functions ";
call RXFUNCADD 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
call SysLoadFuncs
call SysCls
SAY
SAY "*************************************"
SAY "*                                   *"
SAY "*     Registering the One's Part    *"
SAY "*                                   *"
SAY "*************************************"

/* Check to see if the functions are already loaded and */
/* they are unload them if                              */
rc = RXFUNCQUERY('ODLoadOpenDocFuncs')
SAY " REXX function Query on UnloadFuncs rc is " rc
rc = RXFUNCQUERY('ODLoadOpenDocFuncs');
SAY " rc = " rc;
IF rc \= 0 then DO
   SAY " Adding ODLoadOpenDocFuncs"
   rc = RXFUNCADD('ODLoadOpenDocFuncs', 'OPENDOC', 'ODLoadOpenDocFuncs');
   SAY " RXFUNCADD rc for ODLoadOpenDocFuncs = " rc
   IF rc <  0 THEN DO
      SAY " Unable to add  OpenDoc functions, EXITING";
      RETURN;
   END
END
call ODLoadOpenDocFuncs

/*******************************************************/
/*   Make one's changes to the following two lines     */
/*******************************************************/
```

```
className = "YourPart"
dllName    = "yourdll"
cTemplate = "TRUE"
somIRName = ""
SAY "Registering the One's Part"
rc = ODRegisterPartHandlerClass( className, dllName, cTemplate, somIRName)

/* rc of 0 = successfully added, 4 = successfully replaced */
IF rc \=  0 & rc \= 4 then do
   call ODUnLoadOpenDocFuncs
   call RxFuncDrop 'ODLoadOpenDocFuncs'
   SAY " Registration failed, rc = " rc;
   return;
END

call ODUnLoadOpenDocFuncs
call RxFuncDrop 'ODLoadOpenDocFuncs'
SAY "YourPart successfully registered";
RETURN
```

After the part is registered, the part handler's DLL can be replaced as many times as needed without having to reregister unless, of course, the code in the five registration methods has been changed.

Public Utilities

Complete C++ source is provided for the public utilities. This allows the part programmer to use them with any C++ compiler, extend or modify them, and port them to other languages. The source code is hoped to be self-documenting.

The source for the public utilities can be found in the PUBUTILS subdirectory. A makefile is provided to build the utilities into a library and the sample parts link with this library.

The following public utilities are provided:

- ALTPOINT provides C++ class versions of the standard OpenDoc types of ODPoint and ODRect.
- ALTPOLY provides C++ class versions of the standard OpenDoc types of ODPolygon and ODContour. It also contains a helper class PolyEdgeIterator.
- EXCEPT provides macros, functions and a C++ class that help with the implementation of exceptions. Note that EXCEPT.H MUST be included before OS2.H. This utility is documented in the recipe Exception Handling.
- FOCUSLIB provides C++ class that reduces the amount of code that needs to be written in the ODPart::Draw method. Focuslib provides methods that setup and cleanup a GPI HPS using the shape and clipping information from OpenDoc.
- LINEOPS provides geometric operations on lines in 2-D space.
- ODDEBUG provides debugging utilities, for assertions, safe type-casting and logging.
- ODUTILS provides macros and functions for use particularly with reference-counted objects.
- TEMPOBJ provides C++ template classes for exception-safe temporary object references. TempObj and TempRef are simple template classes that act as a transparent wrapper around an OpenDoc object pointer. The temp object can be used wherever a pointer to the object would be used. When the temp object goes out of scope, the object it wraps will be deleted (in the case of TempObj) or released (in the case of TempRef). This includes the case where an exception is thrown; the wrapper is exception-safe. This utility is documented in the recipe Temporary References and Objects.

Exception Handling

The Except utility library facilitates the use of C++ exceptions with OpenDoc and SOM. To use this utility, you must link the file Except.CPP into your library, and include the header Except.H in your source files before the headers of any SOM classes.

The use of this utility is optional; however, if you do not use it you must check the environment (*ev* parameter after every SOM call you make and handle it appropriately, propagating the error back to your caller. The *System Object Model Guide and Reference* manual describes this in more detail.

In the C++ catch/throw exception system, an error is signaled by being thrown. The stack unwinds back to the point where a calling function set up an error handler, and this handler then catches the error. The handler can then do whatever clean-up is necessary. It can then continue on, or (more commonly) reraise the error, which throws it back to the next exception handler on the stack.

If you are already experienced with this style of exception handling, you may want to skim through the following sections until "Exceptions and SOM" which describes features unique to this exception package.

Sample Code

Here is an example of the most basic use of the exception macros; we are ignoring its interaction with SOM for now.

```
void * MyAlloc(ODSize size)
{
  void *p=malloc(size);
  THROW_IF_NULL(p);
}
void foo()
{
  void *p1, *p2;
  p1=MyAlloc(10000);
  TRY
  {
    p2=MyAlloc(10000);
  }
  CATCH_ALL
  {
    free(p1);
    RERAISE;
  }
ENDTRY
  . . .
}
```

This example shows a fail-safe allocation of two pointers. MyAlloc allocates memory and throws an exception if the memory allocation fails. (THROW_IF_NULL is part of the exception macros; it throws an exception if the value passed in is zero). If the first call to MyAlloc fails, it throws an error. Since foo has not set up an exception handler (a TRY ... CATCH_ALL ... ENDTRY block) around that call, the exception will be thrown out of foo into its caller, and so on up the stack until an exception handler is found.

If the first call succeeds, we come to the second call to MyAlloc, which is inside an exception handler. If this call throws an exception, it will be caught by the exception handler, and the block after CATCH_ALL will run. The purpose of this block is to clean up by freeing the p1 pointer, preventing a memory leak. After p1 is freed, it calls RERAISE, which reraises the same exception, which then goes up the stack until the next enclosing exception handler is found. (If we had not called RERAISE, we would have fallen out of the exception handler to the statement after ENDTRY).

If the second call to MyAlloc succeeds, we fall out of the entire exception handler, skipping the CATCH_ALL block entirely (since there was no exception) and ending up at the statement immediately following ENDTRY.

Raising Exceptions

An exception is thrown by calling THROW or one of its variants. This immediately jumps to the most recently installed exception handler, as described above. If Logging is enabled and your code was compiled with the identifier ODDebug defined to 1, the location of the THROW in your source code will be logged along with any message you provide.

THROW(ODError error)
Throws the exception of the ODNativeException class whose value is given by error. There is a list of standard OpenDoc error codes in ErrorDef.XH; platform/OS errors can also be thrown where there is no OpenDoc equivalent. Error must be nonzero; it does not make any sense to throw a kODNoError exception!

THROW_IF_NULL(void* ptr)
Throws the exception of the ODNativeException class whose value is kODErrOutOfMemory if a null pointer is input. This function is intended to be called after you call a memory allocator (such as SOMNew or malloc) which returns NULL if there is not enough memory available.

It should not be used with a function that may return NULL for other reasons. For instance, the PM functions may return NULL if an error occurred; after calling it you should call WinGetLastError to find the actual error code.

THROW_IF_ERROR(ODError error)
Throws an exception of the ODNativeException class if error is non-zero (not equal to kODNoError). Otherwise nothing happens. This is a useful call to wrap around a function call that returns an error code; an example is:

```
THROW_IF_ERROR(DosOpen(...));
```

DosOpen returns zero if it succeeds, otherwise a nonzero error code. Passing the result to THROW_IF_ERROR ensures that the right exception will be thrown if the call failed.

THROW_M(ODError error, const char * message)
It is the same as THROW except that it adds the message to the exception.

THROW_IF_NULL_M(void *ptr, const char * message)
It is the same as THROW_IF_NULL except that it adds the message to the exception.

THROW_IF_ERROR_M(ODError error, const char * message) It is the same as THROW_IF_ERROR that it adds the message to the exception.

Exception Handlers

An exception handler for OpenDoc exceptions consists of a TRY block followed by a CATCH_ALL block, and ends with an ENDTRY:

```
TRY
{
  statements
}
CATCH_ALL
{
  statements
}
ENDTRY
```

(The curly braces are not strictly necessary but are encouraged for readability, and for convenience when using editors that know how to find matching braces for you).

TRY
The statements are executed. If one of the statements, or any function one of the statements calls, throws an OpenDoc exception that reaches this exception handler, then the following CATCH_ALL block will be executed. Otherwise, after the last statement finishes, control passes to the statement following the ENDTRY.

CATCH_ALL
It catches an exception of the ODNativeException class. This class is defined in EXCEPT.H and is thrown by THROW and its variants. If you plan to call a C++ library which may throw other exception classes, you should use the normal C++ try and catch statements to obtain these exceptions.

ENDTRY
Indicates the end of an exception handler. After a TRY or CATCH_ALL block finishes without throwing or reraising an exception, the control passes to the statement following ENDTRY.

It is perfectly legal (and not uncommon) to lexically nest exception handlers in a single function. Any error caught and reraised by the inner handler will be caught by the outer one.

Using exceptions allows you to implement a very useful C++ programming idiom, in which the pair of calls to acquire and release some sort of resource (like a memory block, or a reference to a ref-counted object) are implemented by the constructor and destructor of a stack-based class. This means that you do not have to remember to make the release call since it happens implicitly when the object goes out of scope; when working with exceptions, it means that you can also be confident that, if an exception is thrown, the release will still take place on the way out. This is a lot easier than having to wrap an exception handler around the block.

Exceptions and SOM

The EXCEPT utility adds some special features to simplify working with SOM. The reasons why these features are necessary are:

1. SOM has its own way of returning error codes, based on an environment variable, a pointer to which is passed into every SOM method. Methods you write and methods you call will return errors this way. (For more detail, see the *System Object Model Guide and Reference* manual).

2. You cannot throw a C++ exception, or allow one to be thrown, out of a SOM method. SOM requires that a method return normally, and throwing an exception that is caught by a handler in some other function farther up the stack would violate this.

This implies that the ev parameter must be checked for an error status after every call to a SOM method; and that an exception raised in a SOM method or any function it calls must be caught and its error code stored in the ev parameter. Fortunately, the exception package includes utilities to make this fairly painless. You just need to use variants of the previously defined exception handler macros:

```
SOM_TRY
  SOM_CATCH_ALL
SOM_ENDTRY
```

These macros are identical to the previously defined ones, except that, when they catch an exception of the ODNativeException class, they store the exception value in the method's *ev* parameter (it must be named *ev*) when the catch block goes out of scope.

Since you cannot throw out of a SOM method, it is illegal to use RERAISE in the SOM_CATCH_ALL block. You should exit from the function normally, by falling off the end or calling return.

Here is an example:

```
Baz* ODFooBarNewBaz(ODFooBar *somSelf, Environment *ev, ODULong size)
{
  Baz *baz;
  SOM_TRY
    baz = MyAlloc(sizeof(Baz)+size);
    baz->owner = somSelf;
    baz->size = size;
    SOM_CATCH_ALL
      baz = kODNULL;
  SOM_ENDTRY
  return baz;
}
```

The example shows the implementation of the NewBaz method of the class ODFooBar. Since this method calls MyAlloc, which throws an exception if it fails, it needs an exception handler. If MyAlloc succeeds, the newly allocated Baz structure is initialized and returned. If, on the other hand, MyAlloc throws an exception-whose error code will be kODErrOutOfMemory-the exception handler will catch the exception, and transfer the control to the SOM_CATCH_ALL block. This sets baz to NULL so that NULL will be returned by the return statement. The exception code will be set into the *ev* parameter when the SOM_CATCH_ALL block goes out of scope. (The caller will probably ignore the return value upon seeing that an exception is set in the ev parameter, but it is best to return something safe anyway).

Automatic Environment Checking

Including the header Except.H defines a special preprocessor symbol that modifies the way SOM messages are sent. Any SOM headers (.XH files, for C++) compiled with the *-mchkexcept* option of the SOM compiler, which are included after EXCEPT.H will be modified so that, after the message is sent and control returns to the caller, the environment (ev) is checked and an exception raised of the ODNativeException class if the method returned an error.

Note: The .XH files that are shipped with the Toolkit were compiled with the *-mchkexcept* option.

For example, here is a code snippet that does not use this functionality:

```
#include <ODFoo.xh>
  .
  .
  .
void AFunction(Environment *ev, ODFoo *foo)
{
  long result = foo->Bazz(ev);
  if( ev->_major )
```

```
  {
    // Oops, ODFoo::Bazz returned an error
    result = 0;
    goto handle_error;
  }
  .
  .
  .
  handle_error:
  return result;
}
```

In this example, Except.H is not included, so environment checking is not automatic, and the caller (AFunction) can and must check the environment after every call.

Here is the same example with automatic environment checking:

```
// Enables automatic ev checking for ODFoo!
#include <Except.h>
#include <ODFoo.xh>
  .
  .
  .
long AFunction(Environment *ev, ODFoo *foo)
{
  long result;
  SOM_TRY
    long result = foo->Bazz(ev);
    .
    .
    .
    SOM_CATCH_ALL
      result = kODNULL;
    SOM_ENDTRY
    return result;
}
```

Since Except.H is included before ODFoo.XH, environment-checking code is added to the call to ODFoo's Bazz method. If Bazz encounters an error and returns error status in ev, an exception with that same error code is thrown, which will be caught by the exception handler, and in its turn returned in the ev parameter.

There are two important caveats to keep in mind:

1. It is important that you include Except.H before any headers that declare SOM classes (.XH files, in C++) if you want to use automatic environment checking. It will not take effect for any SOM classes that are declared before Except.H is included.

2. When using automatic environment checking, any SOM method call may throw an exception, so any SOM method that calls other SOM methods must be prepared to handle these exceptions, probably via a SOM_CATCH_ALL.

Temporary References and Objects

When writing OpenDoc-based code, one often needs to create temporary objects that need to be freed, or acquire temporary references to objects that then need to be released. There are two pitfalls with this-the obvious one is that you have to remember to call delete or release on every temporary object. The not so obvious one is that, if you are using the Except utility and an exception is thrown while a temporary is active, the rug will be pulled out from under you and you will not be able to issue the delete or release call. This results in a memory or ref-count leak, unless you put in an exception handler whose job is to clean up these temporaries, which complicates your code and introduces even more room for subtle errors.

Here is an example of code that uses a temporary reference:

```
{
  ODShape *s = frame->GetFrameShape(ev,kODNULL);
  DoSomethingTo(s);
  s->Release(ev);
}
```

We had to remember to call Release on s after we were through with it. And we still have the problem that, if DoSomething throws an

exception, s will be left dangling. We could make the code exception-safe by rewriting it as:

```
{
  ODShape *s = frame->GetFrameShape(ev,kODNULL);
  TRY
    DoSomethingTo(s);
    CATCH_ALL
      s->Release(ev);
      RERAISE;
  ENDTRY
  s->Release(ev);
}
```

This makes the code more complicated, and you have to repeat the Release call twice. It is easy to get this wrong, resulting in code that works fine unless an exception is thrown, in which case it does the wrong thing. Since exceptions happen rarely in normal use, this results in spurious bugs.

An elegant solution to the problem is to make use of stack-based C++ objects whose destructors will be called whenever they go out of scope, whether through exiting a block normally, or via an exception.

Using the TempObj utility, which can be found in the PUBUTILS directory of the Toolkit, the routine can be rewritten as:

```
{
  TempODShape s = frame->GetFrameShape(ev,kODNULL);
  DoSomethingTo(s);
}
```

All we had to do was change ODShape* to TempODShape, and take out the Release call. The rest is the same. Although s is now an actual (stack-based) object, not a pointer, it can still be used as though it were an ODShape*. In particular, the following kinds of things are legal and do what you would expect:

```
s->GetBoundingBox( . . . )
xform->TransformShape(ev,s);
if (s != kODNULL)  . . .
if (s)  . . .

// Note that this does not release the shape s used to point to!
s = frame->GetUsedShape(ev,kODNULL);
s = kODNULL;
```

The Release happens automatically when the block exits or if DoSomething throws an exception. Of course, if s holds a pointer to kODNULL, no Release operation will occur.

Using TempObjs and TempRefs

To use this facility, just #include <TempObj.H> in your source files. This gives you access to the following classes:

TempODFocusSetIterator
TempODFrame
TempODFrameFacetIterator
TempODPart
TempODShape
TempODStorageUnit
TempODTransform
TempODWindow

(Note that iterators are not ref-counted, so the Temp_Iterator classes delete the iterator object at the end instead of releasing it).

Pitfalls

The biggest mistake you can make in using this utility is forgetting that the object is always released. This can bite you if you need to use the

object as the return value of a function:

```
ODShape *foo(ODFrame *frame)
{
  TempODShape s = frame->GetFrameShape(ev,kODNULL);
  DoSomething(s);
  return s;
}
```

The ODShape is going to be released before it is returned, when the destructor of s is called. This is bad news, since the function will return either a pointer to a deleted object, or to an object whose ref-count is one too low. Either case is likely to cause a crash. It is still nice to use a TempODShape in this function, for safety in case DoSomething throws an exception. We just want to tell s not to release itself when it is being returned. You can do this by setting the shape to kODNULL before returning it:

```
ODShape *foo(ODFrame *frame)
{
  TempODShape s = frame->GetFrameShape(ev,kODNULL);
  DoSomething(s);
  ODShape *temp = s;
  // s will not be released by the destructor now
  s = kODNULL;
  return temp;
}
```

Of course, this is a kludge in that we have to store the value of s in a temporary to keep it from being lost. But there is a convenience method called DontRelease that will set the reference to NULL but return its previous value:

```
ODShape *foo(ODFrame *frame)
{
  TempODShape s = frame->GetFrameShape(ev,kODNULL);
  DoSomething(s);
  // Note that we used  ".", not  "->"
  return s.DontDelete();
}
```

Adding New Classes

There are other classes you might want to have Temp____ objects available for.

You can declare a temporary reference to any type of ref-counted object by using the class TempRef<classname>. For instance:

```
TempRef<ODDraft> su = doc->AcquireDraft(ev);
```

You can also use temporary instances of non-ref-counted objects by using TempObj<class>:

```
TempObj<ODPeanutIterator> iter = peanut->GetIterator(ev);
```

Note: Since this defines a template class, you may need to use special compiler options for generating templates. Refer to your compiler documentation.

If, after adding a new class, you get a type-mismatch error in TempObj.H. This may indicate that you are trying to use TempObj with a class that is not a subclass of ODObject, or TempRef with a class that is not a subclass of ODRefCntObject. This can happen even if the class is correct, if the compiler has not seen the declaration of the class before the declaration of the temporary. In other words, the following is wrong:

```
class ODFoo;
TempRef<ODFoo> ref = . . .;
#include "ODFoo.h"
```

At the time that the compiler instantiates the template for ODFoo, it does not know anything about the class, such as whether it is a subclass

of ODRefCntObject, and will therefore give you type-check errors. You can avoid this by including the header for ODFoo before using the TempRef class.

Shell Plug-In

OpenDoc provides users the ability to customize their OpenDoc run-time by installing OpenDoc Shell Plug-Ins. An OpenDoc Shell Plug-In is simply a Dynamic Link Library (DLL) located in the OpenDoc Shell Plug-Ins folder that exports a symbol called ODShellPlugInInstall. This document provides sample code for implementing a Shell Plug-In.

Elements

1. Create a source file with an entry point called ODShellPlugInInstall. If using C++, be sure to type it `extern C` so that the right calling conventions will be used.
2. Set up your build system to produce a DLL from the source file, and to export the symbol ODShellPlugInInstall in the module definition file. Compile and link the library.

Installation

Once you have successfully built the DLL, place it in the OpenDoc Shell Plug-Ins folder. The ID string of the folder is <OD_SHELLPLUGINS>.

Sample Code

```
#define  INCL_DOSERRORS
#define  INCL_ODAPI
#define  INCL_ODDRAFT
#include <OS2.H>
#include "ShPlugIn.h"

#ifdef __cplusplus
extern "C"
{
#endif

APIRET APIENTRY ODShellPluginInstall(Environment* ev,
                                     ODDraft* draft,
                                     ODShellPluginActionCodes* action);

#ifdef __cplusplus
}
#endif

APIRET APIENTRY ODShellPluginInstall(Environment* ev,
                                     ODDraft* draft,
                                     ODShellPluginActionCodes* action)
{
  somPrintf("\pPlugin got called");

  // Now tell the Shell to free our library by calling DosFreeModule
  // as soon as we return.
  // This is not what most plugins will want to do, but it is the right
```

```
  // thing to do if you are just displaying a debug string.
  // If you wanted to leave the library open, say because you would
  // registered an object in an object name space, you would clear the
  // kODShellPluginCloseConnection bit if paranoid, or otherwise just
  // leave *action alone.
  *action |= kODShellPluginCloseConnection;

  // Return NO_ERROR if you want the Shell to continue starting up.
  // Any other error causes the Shell to abort with a dialog telling the user what
  // plugin it was attempting to load when the error occurred and suggesting
  // that he remove it from the plugins folder before attempting to open
  // the document again.
  // Note that the Shell ignores errors returned via the Environment*
  // parameter.
  // Pass ev to OpenDoc routines that you call from here if they require it,
  // but if any of them returns an error that way you will have to deal
  // with it yourself (if it needs dealing with at all).
  // If the Shell gets NO_ERROR back from your plugin it assumes all is
  // well and continues.
  return NO_ERROR;
}
```

Note: When you link the shell plug-in, you must export the ODShellPluginInstall function in the module definition file.

See the SMPDSPM sample shell plug-in in the toolkit.

Notices

Edition Notices

August 1996

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM reseller or IBM marketing representative.

Copyright Notices

Disclaimers

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594
U.S.A.

Asia-Pacific users can inquire, in writing, to the IBM Director of Intellectual Property and Licensing, IBM World Trade Asia Corporation, 2-31 Roppongi 3-chome, Minato-ku, Tokyo 106, Japan.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Department LZKS, 11400 Burnet Road, Austin, TX 78758 U.S.A. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

IBM	OS/2

The following terms are trademarks of other companies:

Apple	Apple Computer, Inc.
C++	American Telephone and Telegraph Company
CORBA	Object Management Group, Inc.
Macintosh	Apple Computer, Inc.
Microsoft	Microsoft Corporation
OpenDoc	Apple Computer, Inc.
PostScript	Adobe Systems, Inc.
UNIX	X/Open Company Limited
Windows	Microsoft Corporation

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

Glossary

This glossary defines many of the terms used in this book. It includes terms and definitions from the *IBM Dictionary of Computing* , as well as terms specific to the OS/2 operating system and the Presentation Manager. It is not a complete glossary for the entire OS/2 operating system; nor is it a complete dictionary of computer terms.

Other primary sources for these definitions are:

- The *American National Standard Dictionary for Information Systems* , ANSI X3.172-1990, copyrighted 1990 by the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. These definitions are identified by the symbol (A) after the definition.
- The *Information Technology Vocabulary* , developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol (I) after the definition; definitions taken from draft international standards,

committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol (T) after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

Glossary Listing

Select a starting letter of glossary terms:

Glossary - 0-9

8.3 file-name format -A file-naming convention in which file names are limited to eight characters before and three characters after a single dot. Usually pronounced "eight-dot-three." See also *non-8.3 file-name format* .

Glossary - A

abstract class -A class used only to derive other classes. An abstract class is never instantiated. Contrast with *concrete class* .

abstract superclass -A superclass listed in the *OSA Event Registry: Standard Suite*s , such as cObject or cOpenableObject, that is used only in definitions of object classes and not for real OSA event objects. See also *object class* .

accelerator -In SAA Common User Access architecture, a key or combination of keys that invokes an application-defined function.

accelerator table -A table used to define which key strokes are treated as *accelerator*s and the commands they are translated into.

access mode -The manner in which an application gains access to a file it has opened. Examples of access modes are read-only, write-only, and read/write.

access permission -All access rights that a user has regarding an object. (I)

action -One of a set of defined tasks that a computer performs. Users request the application to perform an action in several ways, such as typing a command, pressing a function key, or selecting the action name from a menu bar or menu.

action data -Information stored in the undo object's action history that allows a part to reverse the effects of an undoable action.

action history -The cumulative set of reversible actions available at any one time, maintained by the undo object.

action subhistory -A subset of action data added to the undo object's action history by a part in a modal state. The part can then remove the subhistory from the action history without affecting earlier actions.

action type -A constant that defines whether an undoable action is a single-stage action (such as a cut) or part of a two-stage action (such as a drag-move).

activate -(1) For a part, to make ready to receive the selection focus. A frame is activated when a mouse-down event occurs within it. (2) For a window, to bring it to the front by passing the cursor over it.

active frame -The frame that has the selection focus and usually the keyboard focus. Editing takes place in the active frame; the selection or insertion point is displayed within the frame. The active frame usually has the keystroke and menu foci, also.

active part -The part displayed in the active frame. The active part controls the part-specific palettes and menus, and its content contains the selection or insertion point. The active part can be displayed in one or more frames, only one of which is the active frame.

active program -A program currently running on the computer. An active program can be interactive (running and receiving input from the user) or noninteractive (running but not receiving input from the user). See also *interactive program* and *noninteractive program* .

active shape -A shape that describes the portion of a facet within which a part expects to receive user events. If, for example, an embedded part's used shape and active shape are identical, the containing part both draws and accepts events in the unused areas within the embedded part's frame.

active window -The window with which the user is currently interacting.

additional parameter -A keyword-specified descriptor record that a server application uses in addition to the data specified in the direct parameter. For example, an OSA event for arithmetic operations may include additional parameters that specify operands in an equation. Additional parameters may be required, or they may be optional.

address descriptor record -A descriptor record of data type AEAddressDesc that contains the address of the target or source of an OSA event.

address space -(1) The range of addresses available to a program. (A) (2) The area of virtual storage available for a particular job.

alphanumeric video output -Output to the logical video buffer when the video adapter is in text mode and the logical video buffer is addressed by an application as a rectangular array of character cells.

AE record -A descriptor record of data type AERecord that usually contains a list of parameters for an OSA event. See also *OSA event parameter* .

American National Standard Code for Information Interchange -The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (A)

Note: IBM has defined an extension to ASCII code (characters 128-255).

ancestor -See *superclass* .

anchor -A window procedure that handles Presentation Manager* message conversions between an icon procedure and an application.

anchor block -An area of Presentation-Manager-internal resources to allocated process or thread that calls WinInitialize.

anchor point -A point in a window used by a program designer or by a window manager to position a subsequently appearing window.

annotation -A property in a part's storage unit that is separate from the part's contents.

ANSI -American National Standards Institute.

APA -All points addressable.

API -Application programming interface.

application -A collection of software components used to perform specific types of user-oriented work on a computer; for example, a payroll application, an airline reservation application, a network application. See also *conventional application* .

application-modal -Pertaining to a message box or dialog box for which processing must be completed before further interaction with any other window owned by the same application may take place.

application object -In SAA Advanced Common User Access architecture, a form that an application provides for a user; for example, a spreadsheet form. Contrast with *user object* .

application programming interface (API) -A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program.

application result handler -A result handler that is associated with a particular application. Contrast with *system result handler* .

arbitrator -An OpenDoc object that manages negotiation among parts about ownership of shared resources. Examples of such resources are the menu focus, selection focus, keystroke focus, and the serial ports.

area -In computer graphics, a filled shape such as a solid rectangle.

ASCII -American National Standard Code for Information Interchange.

ASCIIZ -A string of ASCII characters that is terminated with a byte containing the value 0.

aspect ratio -In computer graphics, the width-to-height ratio of an area, symbol, or shape.

asynchronous (ASYNC) -(1) Pertaining to two or more processes that do not depend upon the occurrence of specific events such as common timing signals. (T) (2) Without regular time relationship; unexpected or unpredictable with respect to the execution of program instructions. See also *synchronous* .

atom -A constant that represents a string. As soon as a string has been defined as an atom, the atom can be used in place of the string to save space. Strings are associated with their respective atoms in an *atom table* . See *integer atom* .

atom table -A table used to relate *atoms* with the strings that they represent. Also in the table is the mechanism by which the presence of a string can be checked.

atomic operation -An operation that completes its work on an object before another operation can be performed on the same object.

attribute -A characteristic or property that can be controlled, usually to obtain a required appearance; for example, the color of a line. See also *graphics attributes* and *segment attributes* .

automatic link -In Information Presentation Facility (IPF), a link that begins a chain reaction at the primary window. When the user selects the primary window, an automatic link is activated to display secondary windows.

auxiliary storage unit -An extra storage unit that a part uses to store its contents. Contrast with *main storage unit* .

AVIO -Advanced Video Input/Output.

Glossary - B

background -(1) In multiprogramming, the conditions under which low-priority programs are executed. Contrast with *foreground* . (2) An active session that is not currently displayed on the screen.

background color -The color in which the background of a graphic primitive is drawn.

background mix -An attribute that determines how the background of a graphic primitive is combined with the existing color of the graphics presentation space. Contrast with *mix* .

background program -In multiprogramming, a program that executes with a low priority. Contrast with *foreground program* .

base class -See *superclass* .

base draft -The original draft of a document. Every OpenDoc document has a base draft, from which all subsequent drafts are ultimately derived. See also *current draft* .

base menu bar -The menu bar that contains the menus shared by all parts in a document. The document shell installs the base menu bar; parts add their own menus and items.

base object -The object whose interface is extended by an extension object.

Bento -A container suite that implements OpenDoc storage on OS/2 and some other platforms.

Bézier curve -(1) A mathematical technique of specifying smooth continuous lines and surfaces, which require a starting point and a finishing point with several intermediate points that influence or control the path of the linking curve. Named after Dr. P. Bézier. (2) In the AIX Graphics Library, a cubic spline approximation to a set of four control points that passes through the first and fourth control points and that has a continuous slope where two spline segments meet. Named after Dr. P. Bézier.

bias transform -A transform that is applied to measurements in a part's coordinate system to change them into *platform-normal coordinates* .

binding -(1) In programming, an association between a variable and a value for that variable that holds within a defined scope. The scope may be that of a rule, a function call or a procedure invocation. (2) In OpenDoc, the process of selecting an executable code module based on type information. (3) In SOM, a file enabling a compiler to match a method implementation with its declaration. Also called a header file.

bit map -A representation in memory of the data displayed on an APA device, usually the screen.

block -(1) A string of data elements recorded or transmitted as a unit. The elements may be characters, words, or logical records. (T) (2) To record data in a block. (3) A collection of contiguous records recorded as a unit. Blocks are separated by interblock gaps and each block

may contain one or more records. (A)

block device -A storage device that performs I/O operations on blocks of data called sec*tor*s . Data on block devices can be randomly accessed. Block devices are designated by a drive letter (for example, **C:**).

blocking mode -A condition set by an application that determines when its threads might block. For example, an application might set the Pipemode parameter for the DosCreateNPipe function so that its threads perform I/O operations to the named pipe block when no data is available.

border -(1) A visual indication (for example, a separator line or a background color) of the boundaries of a window. (2) For OpenDoc, see *frame border* .

boundary determination -An operation used to compute the size of the smallest rectangle that encloses a graphics object on the screen.

boundary objects -The elements, specified in a range descriptor record, that identify the beginning and end of the range. See also *range descriptor record* .

breakpoint -(1) A point in a computer program where execution may be halted. A breakpoint is usually at the beginning of an instruction where halts, caused by external intervention, are convenient for resuming execution. (T) (2) A place in a program, specified by a command or a condition, where the system halts execution and gives control to the workstation user or to a specified program.

broken pipe -When all of the handles that access one end of a pipe have been closed.

bucket -One or more fields in which the result of an operation is kept.

buffer -(1) A portion of storage used to hold input or output data temporarily. (2) To allocate and schedule the use of buffers. (A)

bundled frame -A frame whose contents do not respond to user events. For example, a mouse click within a bundled frame selects but does not activate the frame.

button -A mechanism used to request or initiate an action. See also *barrel button*s , *bezel button*s , *mouse button* , *push button* , and *radio button* .

byte pipe -Pipes that handle data as byte streams. All unnamed pipes are byte pipes. Named pipes can be byte pipes or message pipes. See also *byte stream* .

byte stream -Data that consists of an unbroken stream of bytes.

Glossary - C

cache -A high-speed buffer storage that contains frequently accessed instructions and data; it is used to reduce access time.

cached micro presentation space -A presentation space from a Presentation-Manager-owned store of micro presentation spaces. It can be used for drawing to a window only, and must be returned to the store when the task is complete.

CAD -Computer-Aided Design.

call -(1) The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point. (I) (A) (2) To transfer control to a procedure, program, routine, or subroutine.

calling sequence -A sequence of instructions together with any associated data necessary to execute a call. (T)

Cancel -An action that removes the current window or menu without processing it, and returns the previous window.

canvas -The platform-specific drawing environment on which frames are laid out. Each window or printing device has one drawing canvas. See also *static canvas* , *dynamic canvas* , and *drawing canvas* .

canvas coordinate space -The coordinate space of the canvas upon which a part's content is drawn. It may or may not be equal to *window coordinate space* .

cascaded menu -In the OS/2 operating system, a menu that appears when the arrow to the right of a cascading choice is selected. It contains a set of choices that are related to the cascading choice. Cascaded menus are used to reduce the length of a menu. See also *cascading choice* .

cascading choice -In SAA Common User Access architecture, a choice in a menu that, when selected, produces a cascaded menu containing other choices. An arrow ( ) appears to the right of the cascading choice.

CASE statement -In PM programming, provides the body of a window procedure. There is usually one CASE statement for each message

type supported by an application.

category -See *part category* .

CGA -Color graphics adapter.

chained list -A list in which the data elements may be dispersed but in which each data element contains information for locating the next. (T) Synonymous with *linked list* .

change ID -(1) A number used to identify a particular instance of clipboard contents. (2) A number used to identify a particular instance of link source data.

character -A letter, digit, or other symbol.

character box -In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single character from a character set. See also *character mode* . Contrast with *character cell* .

character cell -The physical, rectangular space in which any single character is displayed on a screen or printer device. Position is addressed by row and column coordinates. Contrast with *character box* .

character code -The means of addressing a character in a character set, sometimes called *code point* .

character device -A device that performs I/O operations on one character at a time. Because character devices view data as a stream of bytes, character-device data cannot be randomly accessed. Character devices include the keyboard, mouse, and printer, and are referred to by name.

character mode -A mode that, in conjunction with the font type, determines the extent to which graphics characters are affected by the character box, shear, and angle attributes.

character set -(1) An ordered set of unique representations called characters; for example, the 26 letters of English alphabet, Boolean 0 and 1, the set of symbols in the Morse code, and the 128 ASCII characters. (A) (2) All the valid characters for a programming language or for a computer system. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print.

check box -In SAA Advanced Common User Access architecture, a square box with associated text that represents a choice. When a user selects a choice, an X appears in the check box to indicate that the choice is in effect. The user can clear the check box by selecting the choice again. Contrast with *radio button* .

check mark - (1) In SAA Advanced Common User Access architecture, a symbol that shows that a choice is currently in effect. (2) The symbol that is used to indicate a selected item on a pull-down menu.

child class -See *subclass* .

child process -In the OS/2 operating system, a process started by another process, which is called the parent process. Contrast with *parent process* .

child window -A window that appears within the border of its parent window (either a primary window or another child window). When the parent window is resized, moved, or destroyed, the child window also is resized, moved, or destroyed; however, the child window can be moved or resized independently from the parent window, within the boundaries of the parent window. Contrast with *parent window* .

choice -(1) An option that can be selected. The choice can be presented as text, as a symbol (number or letter), or as an icon (a pictorial symbol). (2) In SAA Common User Access architecture, an item that a user can select.

chord -(1) To press more than one button on a pointing device while the pointer is within the limits that the user has specified for the operating environment. (2) In graphics, a short line segment whose end points lie on a circle. Chords are a means for producing a circular image from straight lines. The higher the number of chords per circle, the smoother the circular image.

CI Labs -See *Component Integration Laboratories* .

circular link -A configuration of links in which changes to a lin's destination directly or indirectly affect its source.

class -In object-oriented design or programming, a group of objects that share a common definition and that therefore share common properties, operations, and behavior. Members of the group are called instances of the class.

class hierarchy -The structure by which classes are related through inheritance.

class method -In System Object Model, an action that can be performed on a class object. Synonymous with factory method.

class object -In System Object Model, the run-time implementation of a class.

class style -The set of properties that apply to every window in a window class.

client -(1) A functional unit that receives shared services from a server. (T) (2) A user, as in a client process that uses a named pipe or queue that is created and owned by a server process.

client application -An application that uses OSA events to request a service (for example, printing a list of files, checking the spelling of a list

of words, or performing a numeric calculation) from another application (called a server application). These applications can reside on the same local computer or on remote computers connected to a network.

client area -The part of the window, inside the border, that is below the menu bar. It is the user's work space, where a user types information and selects choices from selection fields. In primary windows, it is where an application programmer presents the objects that a user works on.

client program -An application that creates and manipulates instances of classes.

client window -The window in which the application displays output and receives input. This window is located inside the frame window, under the window title bar and any menu bar, and within any scroll bars.

clip limits -The area of the paper that can be reached by a printer or plotter.

clip shape -(1) The structure by which classes are related through inheritance. (2) A shape that defines the limits of drawing within a facet.

clipboard -In SAA Common User Access architecture, an area of computer memory, or storage, that temporarily holds data. Data in the clipboard is available to other applications.

clipboard focus -In OpenDoc, a designation of ownership of access to the clipboard. The part with the clipboard focus can read from and write to the clipboard.

clipping -In computer graphics, removing those parts of a display image that lie outside a given boundary. (I) (A)

clipping area -The area in which the window can paint.

clipping path -A clipping boundary in world-coordinate space.

clock tick -The minimum unit of time that the system tracks. If the system timer currently counts at a rate of X Hz, the system tracks the time every 1/X of a second. Also known as *time tick* .

CLOCK$ -Character-device name reserved for the system clock.

clone -To copy an object and all its referenced objects. When you clone an object, that object plus all other objects to which there is a *strong persistent reference* in the cloned object are copied.

close -For a frame, to remove from memory but not from storage. A closed frame is not permanently removed from its document. Contrast with *remove* .

Code Fragment Manager (CFM) -The portion of system software that manages the run-time loading and dynamic linking of code modules.

code page -An assignment of graphic characters and control-function meanings to all code points.

code point -(1) Synonym for *character code* . (2) A 1-byte code representing one of 256 potential characters.

code segment -An executable section of programming code within a load module.

coercion handler -In the Open Scripting Architecture, a function that converts data from one descriptor type into another.

color dithering -See *dithering* .

color graphics adapter (CGA) -An adapter that simultaneously provides four colors and is supported by all IBM Personal Computer and Personal System/2 models.

command -The name and parameters associated with an action that a program can perform.

command area -An area composed of a command field prompt and a command entry field.

command entry field -An entry field in which users type commands.

command ID -A position-independent identifier for a menu command. See also *synthetic command ID* .

command line -On a display screen, a display line, sometimes at the bottom of the screen, in which only commands can be entered.

command mode -A state of a system or device in which the user can enter commands.

command prompt -A field prompt showing the location of the command entry field in a panel.

Common Object Request Broker Architecture (CORBA) -A standard promulgated by the Object Management Group industry consortium for defining interactions among objects.

Common Programming Interface (CPI) -Definitions of those application development languages and services that have, or are intended to have, implementations on and a high degree of commonality across the SAA environments. One of the three SAA architectural areas. See also *Common User Access architecture* .

Common User Access (CUA) architecture - Guidelines for the dialog between a human and a workstation or terminal. One of the three SAA

architectural areas. See also *Common Programming Interface* .

comparison descriptor record -A coerced AE record of type typeCompDescriptor that specifies an OSA event object and either another OSA event object or data for the OSA Event Manager to compare to the first object.

compile -To translate a program written in a higher-level

compiled script file -A script file with the file type 'scpt' that contains script data as a resource of type 'scpt'. Before executing the script in a compiled script file, a user must first open the script from an application such as Script Editor.

component -(1) Hardware or software that is part of a functional unit. A functional part of an operating system; for example, the scheduler or supervisor. (2) A set of modules that performs a major function within a system; for example, a compiler or a master scheduler. (3) A software product that functions in the OpenDoc environment. Part editors, part viewers, and services are examples of components. See also *application component, service component* .

Component Integration Laboratories (CI Labs) -A consortium of platform and application vendors that oversees the development and distribution of OpenDoc technology.

component-specific storage descriptor record -A descriptor record returned by OSAStore. The descriptor type for a component-specific storage descriptor record is the scripting component subtype value for the scripting component that created the script data.

composite window -A window composed of other windows (such as a frame window, frame-control windows, and a client window) that are kept together as a unit and that interact with each other.

compound document -A single document containing multiple, heterogeneous data types, each created, presented and edited by its own software. A compound document is made up of *parts* .

computer-aided design (CAD) -The use of a computer to design or change a product, tool, or machine, such as using a computer for drafting or illustrating.

COM1, COM2, COM3 -Character-device names reserved for serial ports 1 through 3.

CON -Character-device name reserved for the console keyboard and screen.

concrete class -A class designed to be instantiated. Contrast with *abstract class* .

conditional cascaded menu -A pull-down menu associated with a menu item that has a cascade mini-push button beside it in an object's pop-up menu. The conditional cascaded menu is displayed when the user selects the mini-push button.

connect or reconnect -For a frame object, to reestablish its connection to the part it displays. Reconnecting a frame may involve recreating it from storage.

container -(1) In SAA Common User Access architecture, an object that holds other objects. A folder is an example of a container object. (2) A holder of persistent data (documents); part of the OpenDoc *container suite* . (3) An OSA event object that contains another OSA event object. A container is specified in an object specifier record by a keyword-specified descriptor record with the keyword keyAEContainer. The keyword-specified descriptor record is usually another object specifier record. It can also be a null descriptor record, or it can be used much like a variable when the OSA Event Manager determines a range or performs a series of tests. The objects a container contains can be either elements or properties. (See also *OSA event object* , *element* , *object specifier record* , *property* , *container part* , *folder* and *object* .

container hierarchy -The chain of containers that determine the location of one or more OSA event objects. See also *container* .

container part -A part that can embed other parts within its content. A container part is capable of being a *containing part* . Contrast with *simple part* and *noncontainer part* . See also *container application* .

containing frame -The display frame of an embedded frame's containing part. Each embedded frame has one containing frame; each containing frame has one or more embedded frames.

containing part -The part that immediately contains an embedded part. Each embedded part has one containing part; each containing part has one or more embedded parts.

container suite -A document storage architecture, built on top of a platform's native file system, that allows for the creation, storage, and retrieval of compound documents. A container suite is implemented as a set of OpenDoc classes: containers, documents, drafts, and storage units. See also *Bento* .

containment -A relationship between objects wherein an object of one class contains a reference to an object of another class. Contrast with *inheritance* .

content -See *part content* .

content area -The potentially visible area of a part as viewed in a frame or window. If the content area is greater than the area of the frame or window, only a portion of the part can be viewed at a time.

content coordinate space -The coordinate space defined by applying the internal transform of a frame to a point in *frame coordinate space* .

content element -A data item that can be seen by the user and is presented by a part's content model. Content elements can be manipulated through the graphical or scripting interface to a part.

content extent -The vertical dimension of the content area of a part in a frame. Content extent is used to calculate *bias transforms* .

content model -The specification of a part's contents (the data types of its content elements) and its content operations (the actions that can be performed on it and the interactions among its content elements).

content object -A content element that can be represented as an object and thus accessed and manipulated through semantic events.

content operation -A user action that manipulates a content element.

content property -A visual or behavioral characteristic of a containing part, such as its text font, that it makes available for embedded parts to adopt. Embedded parts can adopt the content properties of their containing parts, thus giving a more uniform appearance to a set of parts. Contrast with *property* and *Info property* .

content storage unit -The main storage unit of the Clipboard, drag-and-drop object, link source object, or link object.

content transform -The composite transform that converts from a part's content coordinates to its canvas coordinates.

content view type -See *frame view type* .

context -The outermost object in the object hierarchy defined by the direct parameter of an OSA event.

contextual help -In SAA Common User Access Architecture, help that gives specific information about the item the cursor is on. The help is contextual because it provides information about a specific item as it is currently being used. Contrast with *extended help* .

contiguous -Touching or joining at a common edge or boundary, for example, an unbroken consecutive series of storage locations.

control -In SAA Advanced Common User Access architecture, a component of the user interface that allows a user to select choices or type information; for example, a check box, an entry field, a radio button.

control area -A storage area used by a computer program to hold control information. (I) (A)

Control Program -(1) The basic functions of the operating system, including DOS emulation and the support for keyboard, mouse, and video input/output. (2) A computer program designed to schedule and to supervise the execution of programs of a computer system. (I) (A)

control window -A window that is used as part of a composite window to perform simple input and output tasks. Radio buttons and check boxes are examples.

control word -An instruction within a document that identifies its parts or indicates how to format the document.

conventional application -An application that directly handles events, opens documents, and is wholly responsible for manipulating, storing, and retrieving all of the data in its documents. Contrast with *application component* .

coordinate bias -The difference between a given coordinate system and *platform-normal coordinate*s . Coordinate bias typically involves both a change in axis polarity and an offset.

coordinate space -A two-dimensional set of points used to generate output on a video display of printer.

Copy -A choice that places onto the clipboard, a copy of what the user has selected. See also *Cut* and *Paste* .

CORBA -See *Common Object Request Broker Architecture* .

core OSA event -An OSA event defined as part of the Core suite of OSA events in the *OSA Event Registry: Standard Suite*s .

Core suite -The set of OSA events that any scriptable part is expected to support.

correlation -The action of determining which element or object within a picture is at a given position on the display. This follows a *pick* operation.

coverpage window -A window in which the application's help information is displayed.

CPI -Common Programming Interface.

critical extended attribute -An extended attribute that is necessary for the correct operation of the system or a particular application.

critical section -(1) In programming languages, a part of an asynchronous procedure that cannot be executed simultaneously with a certain part of another asynchronous procedure.

Note: Part of the other asynchronous procedure also is a critical section. (2) A section of code that is not reentrant; that is, code that can be executed by only one thread at a time.

CUA architecture -Common User Access architecture.

current draft -The most recent draft of an OpenDoc document. Only the current draft can be edited.

current frame -During drawing, the frame that is being drawn or within which editing is occurring.

current position -In computer graphics, the position, in user coordinates, that becomes the starting point for the next graphics routine, if that routine does not explicitly specify a starting point.

cursor -A symbol displayed on the screen and associated with an input device. The cursor indicates where input from the device will be placed. Types of cursors include text cursors, graphics cursors, and selection cursors. Contrast with *pointer* and *input focus* .

custom OSA event -An OSA event you define for use by your own applications. Instead of creating custom OSA events, you should try to use the standard OSA events and extend their definitions as necessary for your application. If you think you need to define custom OSA events, you should check with the OSA event Registrar to find out whether OSA events that already exist or are under development can be adapted to the needs of your application.

customizable -A level of scripting support of a part. A customizable part defines content objects and operations for interface elements such as menus and buttons; it allows the user to change its behavior during virtually any user action. Contrast with *scriptable* and *recordable* .

Cut -In SAA Common User Access architecture, a choice that removes a selected object, or a part of an object, to the clipboard, usually compressing the space it occupied in a window. See also *Copy* and *Paste* .

Glossary - D

daisy chain -A method of device interconnection for determining interrupt priority by connecting the interrupt sources serially.

database extension -The interface between the Data Access Manager and a data server.

data segment -A nonexecutable section of a program module; that is, a section of a program that contains data definitions.

data server -An application that acts as an interface between a database extension on a personal computer and a data source, which can be on the personal computer or on a remote host computer. A data server can be a database server program that can provide an interface to a variety of different databases, or it can be the data source itself, such as an application program.

data structure -The syntactic structure of symbolic expressions and their storage-allocation characteristics. (T)

data transfer -The movement of data from one object to another by way of the clipboard or by direct manipulation.

DBCS -Double-byte character set.

DDE -Dynamic data exchange.

deadlock -(1) Unresolved contention for the use of a resource. (2) An error condition in which processing cannot continue because each of two elements of the process is waiting for an action by, or a response from, the other. (3) An impasse that occurs when multiple processes are waiting for the availability of a resource that will not become available because it is being held by another process that is in a similar wait state.

debug -To detect, diagnose, and eliminate errors in programs. (T)

decipoint -In printing, one tenth of a point. There are 72 points in an inch.

default container -The outermost container in an application's container hierarchy; usually the application itself. See also *container hierarchy*.

default editor for kind -A user-specified choice of part editor to use with parts whose *preferred editor* is not present.

default object accessorse -Object accessors provided by OpenDoc that can be used to resolve content objects or properties of parts that do not themselves support scripting. Default accessors can return tokens representing an embedded frame, a standard Info property of a part, or a context switch (swap token).

default procedure -A function provided by the Presentation Manager Interface that may be used to process standard messages from dialogs or windows.

default scripting component -The scripting component used by the generic scripting component when an application passes kOSANullScript rather than a valid script ID to OSACompile or OSAStartRecording.

default value -A value assumed when no value has been specified. Synonymous with assumed value. For example, in the graphics programming interface, the default line-type is 'solid'.

definition list -A type of list that pairs a term and its description.

delta -An application-defined threshold, or number of container items, from either end of the list.

derived class -See *subclass* .

descendant -See *child process* and *subclass* .

descriptive text -Text used in addition to a field prompt to give more information about a field.

descriptor list -A descriptor record of data type AEDescList whose data handle refers to a list of descriptor records.

descriptor record -A data structure of type AEDesc that consists of a handle to data and a descriptor type that identifies the type of the data referred to by the handle. Descriptor records are the fundamental data structures from which OSA events are constructed.

descriptor type -An identifier for the type of data referred to by the handle in a descriptor record.

Deselect all -A choice that cancels the selection of all of the objects that have been selected in that window.

desktop window -The window, corresponding to the physical device, against which all other types of windows are established.

destination content -The content at the destination of a link. It is a copy of the *source content* .

destination part -For a link, the part that displays the information copied from the source of the link. Contrast with *source part* .

detached process -A background process that runs independent of the parent process.

detent -A point on a slider that represents an exact value to which a user can move the slider arm.

device context -A logical description of a data destination such as memory, metafile, display, printer, or plotter. See also *direct device* context, *information device context* , *memory device context* , *metafile device context* , *queued device context* , and *screen device context* .

device driver -A file that contains the code needed to attach and use a device such as a display, printer, or plotter.

device space -(1) Coordinate space in which graphics are assembled after all GPI transformations have been applied. Device space is defined in device-specific units. (2) In computer graphics, a space defined by the complete set of addressable points of a display device. (A)

dialog -The interchange of information between a computer and its user through a sequence of requests by the user and the presentation of responses by the computer.

dialog box -In SAA Advanced Common User Access architecture, a movable window, fixed in size, containing controls that a user uses to provide information required by an application so that it can continue to process a user request. See also *message box, primary window, secondary window* . Also known as a *pop-up window* .

Dialog Box Editor -A *WYSIWYG* editor that creates dialog boxes for communicating with the application user.

dialog item -A component (for example, a menu or a button) of a dialog box. Dialog items are also used when creating dialog templates.

dialog procedure -A dialog window that is controlled by a window procedure. It is responsible for responding to all messages sent to the dialog window.

dialog tag language -A markup language used by the DTL compiler to create dialog objects.

dialog template -The definition of a dialog box, which contains details of its position, appearance, and window ID, and the window ID of each of its child windows.

direct device context -A logical description of a data destination that is a device other than the screen (for example, a printer or plotter), and where the output is not to go through the spooler. Its purpose is to satisfy queries. See also *device context* .

direct manipulation -The user's ability to interact with an object by using the mouse, typically by dragging an object around on the Desktop and dropping it on other objects.

direct memory access (DMA) -A technique for moving data directly between main storage and peripheral equipment without requiring processing of the data by the processing unit.(T)

direct parameter -The parameter in an OSA event that contains the data or object specifier record to be used by the server application. For example, a list of documents to be opened is specified in the direct parameter of the Open Documents event. See also *OSA event parameter* .

directory -A type of file containing the names and controlling information for other files or other directories.

dispatcher -The OpenDoc object that directs user events and semantic events to the correct part.

dispatch module -An OpenDoc object used by the dispatcher to dispatch events of a certain type to part editors.

display frame -A frame in which a part is displayed. A part's display frames are created by and embedded in its containing part. Contrast with *embedded frame* .

display-frames list -A part's list of all the frames in which it is displayed. If a part is displayed in only one frame, it has only one element in this list.

display point -Synonym for *pel* .

display property -A visual characteristic of a containing part, such as its text font, that it makes available for embedded parts to adopt. Embedded parts can adopt the display characteristics of their containing parts that they understand, thus giving a more uniform appearance to a set of parts. Display properties are stored as properties in a storage unit passed from containing part to embedded part.

Distributed SOM (DSOM) -Distributed System Object Model. A version of SOM that provides remote access to SOM objects in a transparent way that insulates client programmers from having to know the location or platform type where a target object will be instantiated. DSOM allows programmers to use the same object model independently of whether the objects they access are in the same process, in another process on the same machine, or across distributed networks.

dithering -(1) The process used in color displays whereby every other pel is set to one color, and the intermediate pels are set to another. Together they produce the effect of a third color at normal viewing distances. This process can only be used on solid areas of color; it does not work, for example, on narrow lines. (2) In computer graphics, a technique of interleaving dark and light pixels so that the resulting image looks smoothly shaded when viewed from a distance.

DMA -Direct memory access.

document -In OpenDoc, a user-organized collection of parts, all stored together.

document part -See *part* .

document process -A thread of execution that runs the document shell program. The document process provides the interface between the operating system and part editors. It accepts events from the operating system, provides the address space into which parts are loaded, and provides access to the window system and other features.

document shell -A program that provides an environment for all the parts in a document. The shell maintains the major document global databases: storage, window state, arbitrator, and dispatcher. This code also provides basic document behavior such as document creation, opening, saving, printing, and closing. OpenDoc provides a default document shell for each platform.

document window -A window that displays an OpenDoc document. The edges of the content area of the window represent the frame border of the document's root part. The OpenDoc document shell manages opening and closing of document windows. Contrast with *part window* .

DOS Protect Mode Interface (DPMI) -An interface between protect mode and real mode programs.

double-byte character set (DBCS) -A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese, and Korean, which contain more characters than can be represented by 256 code points, require double-byte character sets. Since each character requires two bytes, the entering, displaying, and printing of DBCS characters requires hardware and software that can support DBCS.

doubleword -A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. (A)

DPMI -DOS Protect Mode Interface.

draft -A configuration of a document, defined at a certain point in time by the user. A document is made up of a set of drafts.

draft key -A number that identifies a specific cloning transaction.

draft permissions -A specification of the class of read/write access that a part editor has to a draft.

drag -In SAA Common User Access, to use a pointing device to move an object; for example, clicking on a window border, and dragging it to make the window larger.

drag and drop -A facility of OpenDoc that allows users to move or copy data through direct manipulation.

drag-copy -A drag-and-drop operation in which the dragged data remains at the source, and a copy is inserted at the destination.

drag-move -A drag-and-drop operation in which the dragged data is deleted from the source and inserted at the destination.

dragging -(1) In computer graphics, moving an object on the display screen as if it were attached to the pointer. (2) In computer graphics, moving one or more segments on a display surface by translating. (I) (A)

drawing canvas -The platform-specific drawing environment on which frames are laid out. Each window or printing device has one drawing canvas. See also *canvas* , *static canvas* , and *dynamic canvas* .

drawing chain -See *segment chain* .

drop -To fix the position of an object that is being dragged, by releasing the select button of the pointing device. See also *drag* .

DSOM -Distributed System Object Model. A version of SOM that works transparently over a network.

DTL -Dialog tag language.

DTS -Direct-To-SOM.

dual-boot function -A feature of the OS/2 operating system that allows the user to start DOS from within the operating system, or an OS/2 session from within DOS.

duplex -Pertaining to communication in which data can be sent and received at the same time. Synonymous with *full duplex* .

dynamic canvas -A drawing canvas that can potentially be changed, such as a window, that can be scrolled or paged to display different portions of a part's data. Contrast with *static canvas* .

dynamic data exchange (DDE) -A message protocol used to communicate between applications that share data. The protocol uses shared memory as the means of exchanging data between applications.

dynamic data formatting -A formatting procedure that enables you to incorporate text, bit maps or metafiles in an IPF window at execution time.

dynamic link library -A collection of executable programming code and data that is bound to an application at load time or run time, rather than during linking. The programming code and data in a dynamic link library can be shared by several applications simultaneously.

dynamic linking -The process of resolving external references in a program module at load time or run time rather than during linking.

dynamic segments -Graphics segments drawn in exclusive-OR mix mode so that they can be moved from one screen position to another without affecting the rest of the displayed picture.

dynamic storage -(1) A device that stores data in a manner that permits the data to move or vary with time such that the specified data is not always available for recovery. (A) (2) A storage in which the cells require repetitive application of control signals in order to retain stored data. Such repetitive application of the control signals is called a refresh operation. A dynamic storage may use static addressing or sensing circuits. (A) (3) See also *static storage* .

dynamic time slicing -Varies the size of the time slice depending on system load and paging activity.

dynamic-link module -A module that is linked at load time or run time.

Glossary - E

EBCDIC -Extended binary-coded decimal interchange code. A coded character set consisting of 8-bit coded characters (9 bits including parity check), used for information interchange among data processing systems, data communications systems, and associated equipment.

edge-triggered -Pertaining to an event semaphore that is posted then reset before a waiting thread gets a chance to run. The semaphore is considered to be posted for the rest of that thread's waiting period; the thread does not have to wait for the semaphore to be posted again.

edit-in-place -See *in-place editing* .

editor of last resort -The part editor that displays any part for which there is no available part editor on the system. The editor of last resort typically displays a gray rectangle representing the part's frame.

editor properties -A notebook, accessed through the **Edit** menu, in which the user can view and change properties for the part editor of the currently active part.

EGA -Extended graphics adapter.

element -(1) An entry in a graphics segment that comprises one or more graphics orders and that is addressed by the element pointer. (2) An OSA event object contained by another OSA event object specified as the element's container. An OSA event object can contain many elements of the same element class, whereas an OSA event object can have only one of each of its properties. See also *OSA event object* , *container* , *element classes* , *property* .

element classes -In the *OSA Event Registry: Standard Suites* , a list of the object classes for the elements that an OSA event object of a given object class can contain. See also *OSA event object* , *object class* .

embed -To display one part in a frame within another part. The embedded part retains its identity as a separate part from the containing part. Contrast with *incorporate* .

embedded content -Content displayed in an embedded frame. A containing part editor does not directly manipulate embedded content. Contrast with *intrinsic content* .

embedded frame -A frame that displays an embedded part. The embedded frame itself is considered intrinsic content of the containing part; the part displayed within the frame is not.

embedded-frames list -A containing part's private list of all the frames embedded within it.

embedded part -A part displayed in an embedded frame. The data for an embedded part is stored within the same draft as its containing part. An embedded part is copied during a duplication of its containing part. An embedded part may itself be a containing part, unless it is a *noncontainer part* .

embedding part -A part that is capable of embedding other parts within its content; that is, it is capable of being a containing part. See also *container part* . Contrast with *nonembedding part* .

EMS -Expanded Memory Specification.

encapsulation -Hiding an object's implementation, that is, its private, internal data and methods. Private variables and methods are accessible only to the object that contains them.

entry field -In SAA Common User Access architecture, an area where a user types information. Its boundaries are usually indicated. See also s*election field* .

entry-field control -The component of a user interface that provides the means by which the application receives data entered by the user in an entry field. When it has the input focus, the entry field displays a flashing pointer at the position where the next typed character will go.

entry panel -A defined panel type containing one or more entry fields and protected information such as headings, prompts, and explanatory text.

environment parameter -A parameter used by all methods of SOM objects to pass exceptions.

environment segment -The list of environment variables and their values for a process.

environment strings -ASCII text strings that define the value of environment variables.

environment variables -Variables that describe the execution environment of a process. These variables are named by the operating system or by the application. Environment variables named by the operating system are PATH, DPATH, INCLUDE, INIT, LIB, PROMPT, and TEMP. The values of environment variables are defined by the user in the CONFIG.SYS file, or by using the SET command at the OS/2 command prompt.

error callback function -An object callback function that gives the OSA Event Manager an address. The OSA Event Manager writes to this address the descriptor record it is currently working with if an error occurs during the resolution of an object specifier record. See also *object callback function* .

error message -An indication that an error has been detected. (A)

event -See *user event* . Contrast with s*emantic event* .

event class -An attribute that identifies a group of related OSA events. The event class appears in the message field of the OSA event's event record. The event class and the event ID identify the action an OSA event performs. See also *OSA event attribute* , e*vent ID* .

event handler -(1) A routine that executes in response to receiving a user event. (2) See s*emantic-event handler* .

event-info structure -A data structure that carries information about an OpenDoc user event in addition to that provided by the event structure.

event ID -An attribute that identifies a particular OSA event within a group of related OSA events. The event ID appears in the where field of the OSA event's event record. The event ID and the event class identify the action an OSA event performs. See also *OSA event* attribute, e*vent cla*ss .

Event Manager -The collection of routines that an application can use to receive information about actions performed by the user, to receive notice of changes in the processing status of the application, and to communicate with other applications.

event semaphore -A semaphore that enables a thread to signal a waiting thread or threads that an event has occurred or that a task has been completed. The waiting threads can then perform an action that is dependent on the completion of the signaled event.

event structure -A platform-specific structure that carries information about an OpenDoc user event.

exception -In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it.

exclusive focus -A focus that can be owned by only one frame at a time. The selection focus, for example, is exclusive; the user can edit within only one frame at a time. Contrast with *non-exclusive focu*s .

exclusive system semaphore -A system semaphore that can be modified only by threads within the same process.

executable file -(1) A file that contains programs or commands that perform operations or actions to be taken. (2) A collection of related data records that execute programs.

exit -To execute an instruction within a portion of a computer program in order to terminate the execution of that portion. Such portions of computer programs include loops, subroutines, modules, and so on. (T) Repeated exit requests return the user to the point from which all functions provided to the system are accessible. Contrast with *cancel* .

expanded memory specification (EMS) -Enables DOS applications to access memory above the 1MB real mode addressing limit.

extended attribute -An additional piece of information about a file object, such as its data format or category. It consists of a name and a value. A file object may have more than one extended attribute associated with it.

extended-choice selection -A mode that allows the user to select more than one item from a window. Not all windows allow extended choice selection. Contrast with *multiple-choice selection* .

extended help -In SAA Common User Access architecture, a help action that provides information about the contents of the application window from which a user requested help. Contrast with *contextual help* .

extension -An OpenDoc object that extends the programming interface of another OpenDoc object. Part editors, for example, can provide additional interfaces through extensions. An object class that duplicates all the characteristics of an object class of the same name and adds some of its own. Like a word in a dictionary, a single object class ID can have several related definitions.

extent -Continuous space on a disk or diskette that is occupied by or reserved for a particular data set, data space, or file.

external link -In Information Presentation Facility, a link that connects external online document files.

external transform -A transform that is applied to a facet to position, scale, or otherwise transform the facet and the image drawn within it. The external transform loates the facet in the coordinate space of the frame's containing part. Contrast with *internal transform* .

externalize -For a part or other OpenDoc object, to transform its in-memory representation into a persistent form in a storage unit. See also *write* . Contrast with *internalize* .

extracted draft -A draft that is extracted from a document into a new document.

Glossary - F

facet -An object that describes where a frame is displayed on a canvas.

factoring -Using OSA events to separate the code that controls an application's user interface from the code that responds to the user's manipulation of the interface. In a fully factored application, any significant user actions generate OSA events that a scripting component can record as statements in a compiled script. See also *recordable application* .

factory method -A method in one class that creates an instance of another class.

family-mode application -An application program that can run in the OS/2 environment and in the DOS environment; however, it cannot take advantage of many of the OS/2-mode facilities, such as multitasking, interprocess communication, and dynamic linking.

FAT -File allocation table.

FEA -Full extended attribute.

fidelity -The faithfulness of translation attained (or attainable) between data of different part kinds. For a given part kind, other part kinds are ranked in fidelity by the level at which their editors can translate its data without loss.

field-level help -Information specific to the field on which the cursor is positioned. This help function is "contextual" because it provides information about a specific item as it is currently used; the information is dependent upon the context within the work session.

FIFO -First-in-first-out. (A)

file -A named set of records stored or processed as a unit. (T)

file allocation table (FAT) -In IBM personal computers, a table used by the operating system to allocate space on a disk for a file, and to locate and chain together parts of the file that may be scattered on different sectors so that the file can be used in a random or sequential manner.

file attribute -Any of the attributes that describe the characteristics of a file.

file specification -The full identifier for a file, which includes its drive designation, path, file name, and extension.

file system -The combination of software and hardware that supports storing information on a storage device.

file system driver (FSD) -A program that manages file I\O and controls the format of information on the storage media.

fillet -A curve that is tangential to the end points of two adjoining lines. See also *polyfillet* .

filtering -An application process that changes the order of data in a queue.

first-in-first-out (FIFO) -A queuing technique in which the next item to be retrieved is the item that has been in the queue for the longest time. (A)

flag -(1) An indicator or parameter that shows the setting of a switch. (2) A character that signals the occurrence of some condition, such as the end of a word. (A) (3) A characteristic of a file or directory that enables it to be used in certain ways. See also *archive flag* , *hidden flag* , and *read-only flag* .

focus -A designation of ownership of a shared resource such as menus, selection, keystrokes, and serial ports. The part that owns a focus has use of that shared resource.

focus module -An OpenDoc object used by the arbitrator to assign an owner or owners to a given focus type.

focus set -A group of foci requested as a unit.

folder -A container used to organize objects.

font -A particular size and style of typeface that contains definitions of character sets, marker sets, and pattern sets.

Font Editor -A utility program provided with the IBM Developers Toolkit that enables the design and creation of new fonts.

foreground program -(1) The program with which the user is currently interacting. Also known as *interactive program* . Contrast with *background program* . (2) In multiprogramming, a high-priority program.

frame -(1) The part of a window that can contain several different visual elements specified by the application, but drawn and controlled by the Presentation Manager. (2) In OpenDoc, a bounded portion of the content area of a part, defining the location of an embedded part. The edge of a frame marks the boundary between intrinsic content and embedded content. A frame can be a rectangle or any other, even irregular, shape.

frame border -A visual indication of the boundary of a frame. The appearance of the frame border indicates the state of the frame (active, inactive, or selected). The frame border is drawn and manipulated by the containing part or by OpenDoc, not by the part within the frame.

frame coordinate space -The coordinate space in which a part's frame shape, used shape, active shape, and clip shape are defined. Contrast with *content coordinate space* . See also *window coordinate space* , *canvas coordinate space* .

frame group -A set of embedded frames that a containing part designates as related, for purposes such as flowing content from one frame to another. Each frame group has its own *group ID* ; frames within a frame group have a *frame sequence* .

frame negotiation -The process of adjusting the size and shape of an embedded frame. Embedded parts can request changes to their frames, but the containing parts control the changes that occur.

frame sequence -The order of frames in a frame group.

frame shape -A shape that defines a frame and its border, expressed in terms of the frame's local coordinate space.

frame styles -Standard window layouts provided by the Presentation Manager.

frame transform -The composite transform that converts from a part's frame coordinates to its canvas coordinates

frame view type -A view type in which all or a portion of a part's contents is displayed within a frame, the border of which is visible when the part is active or selected. Other possible view types for displaying a part include large icon, small icon, and thumbnail. Frame view type is sometimes called content view type.

FSD -File system driver.

full-duplex -Synonym for *duplex* .

full-screen application -An application that has complete control of the screen.

fully scriptable -Characteristic of a scriptable part in which semantic events can invoke any action a user might be able to perform.

function -(1) In a programming language, a block, with or without formal parameters, whose execution is invoked by means of a call. (2) A set of related control statements that cause one or more programs to be performed.

function key -A key that causes a specified sequence of operations to be performed when it is pressed, for example, F1 and Alt-K.

function key area -The area at the bottom of a window that contains function key assignments such as F1=Help.

functional-area OSA event -A standard OSA event supported by applications with related features; for example, an OSA event related to text manipulation for word-processing applications, or an OSA event related to graphics manipulation for drawing applications. Functional-area OSA events are defined by CI Labs, in consultation with interested developers and are published in the *OSA Event Registry: Standard Suite*s .

Glossary - G

GDT -Global Descriptor Table.

general protection fault -An exception condition that occurs when a process attempts to use storage or a module that has some level of protection assigned to it, such as I/O privilege level. See also *IOPL code segment* .

Global Descriptor Table (GDT) -A table that defines code and data segments available to all tasks in an application.

global dynamic-link module -A dynamic-link module that can be shared by all processes in the system that refer to the module name.

global file-name character -Either a question mark (?) or an asterisk (*) used as a variable in a file name or file name extension when referring to a particular file or group of files.

glyph -A graphic symbol whose appearance conveys information.

GPI -Graphics programming interface.

graphic primitive -In computer graphics, a basic element, such as an arc or a line, that is not made up of smaller parts and that is used to create diagrams and pictures. See also *graphic*s *segment* .

graphics -(1) A picture defined in terms of graphic primitives and graphics attributes. (2) The making of charts and pictures. (3) Pertaining to charts, tables, and their creation. (4) See *computer graphics, coordinate graphics, fixed-image graphics, interactive graphics, passive graphics, raster graphic*s .

graphics attributes -Attributes that apply to graphic primitives. Examples are color, line type, and shading-pattern definition. See also *segment attribute*s .

graphics field -The clipping boundary that defines the visible part of the presentation-page contents.

graphics mode -One of several states of a display. The mode determines the resolution and color content of the screen.

graphics model space -The conceptual coordinate space in which a picture is constructed after any model transforms have been applied. Also known as *model space* .

Graphics programming interface -The formally defined programming language that is between an IBM graphics program and the user of the program.

graphics segment -A sequence of related graphic primitives and graphics attributes. See also *graphic primitive* .

graphics system -A specific drawing architecture. Some graphics systems (such as Display PostScript) are available on more than one platform; some platforms support more than one graphics system.

graying -The indication that a choice on a pull-down is unavailable.

group -A collection of logically connected controls. For example, the buttons controlling paper size for a printer could be called a group. See also *program group* .

group ID -In OpenDoc, a number that identifies a frame group, assigned by the group's containing part.

Glossary - H

handle -(1) An identifier that represents an object, such as a device or window, to the Presentation Interface. (2) In the Advanced DOS and OS/2 operating systems, a binary value created by the system that identifies a drive, directory, and file so that the file can be found and opened.

hard error -An error condition on a network that requires either that the system be reconfigured or that the source of the error be removed before the system can resume reliable operation.

header -(1) System-defined control information that precedes user data. (2) The portion of a message that contains control information for the message, such as one or more destination fields, name of the originating station, input sequence number, character string indicating the type of message, and priority level for the message.

heading tags -A document element that enables information to be displayed in windows, and that controls entries in the contents window controls placement of push buttons in a window, and defines the shape and size of windows.

heap -An area of free storage available for dynamic allocation by an application. Its size varies according to the storage requirements of the application.

help function -(1) A function that provides information about a specific field, an application panel, or information about the help facility. (2) One or more display images that describe how to use application software or how to do a system operation.

Help index -In SAA Common User Access architecture, a help action that provides an index of the help information available for an application.

help panel -A panel with information to assist users that is displayed in response to a help request from the user.

help window -A Common-User-Access-defined secondary window that displays information when the user requests help.

hide button -In the OS/2 operating system, a small, square button located in the right-hand corner of the title bar of a window that, when selected, removes from the screen all the windows associated with that window. Contrast with *maximize button* . See also *restore button* .

hidden file -An operating system file that is not displayed by a directory listing.

hierarchical inheritance -The relationship between parent and child classes. An object that is lower in the inheritance hierarchy than another object, inherits all the characteristics and behaviors of the objects above it in the hierarchy.

hierarchy -A tree of segments beginning with the root segment and proceeding downward to dependent segment types.

high-performance file system (HPFS) -In the OS/2 operating system, an installable file system that uses high-speed buffer storage, known as a cache, to provide fast access to large disk volumes. The file system also supports the coexistence of multiple, active file systems on a single personal computer, with the capability of multiple and different storage devices. File names used with the HPFS can have as many as 254 characters.

hit testing -The means of identifying which window is associated with which input device event.

hook -A point in a system-defined function where an application can supply additional code that the system processes as though it were part of the function.

hook chain -A sequence of hook procedures that are "chained" together so that each event is passed, in turn, to each procedure in the chain.

hot part -A part, such as a control, that performs an action (like running a script), rather than activating itself, when it receives a mouse click.

hot spot -The part of the pointer that must touch an object before it can be selected. This is usually the tip of the pointer. Contrast with *action point* .

HPFS -high-performance file system.

hypergraphic link -A connection between one piece of information and another through the use of graphics.

hypertext -A way of presenting information online with connections between one piece of information and another, called *hypertext links* . See also *hypertext link* .

hypertext link -A connection between one piece of information and another.

Glossary - I

I/O operation -An input operation to, or output operation from a device attached to a computer.

I-beam pointer -A pointer that indicates an area, such as an entry field in which text can be edited.

icon -(1) In SAA Advanced Common User Access architecture, a graphical representation of an object, consisting of an image, image background, and a label. Icons can represent items (such as a document file) that the user wants to work on, and actions that the user wants to perform. In the Presentation Manager, icons are used for data objects, system actions, and minimized programs. (2) In

OpenDoc, a small, type-specific picture with a name. Possible iconic view types for displaying a part include as a (standard) *large icon* , s*mall icon* , or *thumbnail* ; the other possible view type is in a *frame* .

icon area -In the Presentation Manager, the area at the bottom of the screen that is normally used to display the icons for minimized programs.

Icon Editor -The Presentation Manager-provided tool for creating icons.

identity transform -A transform that has no effect on points to which it is applied.

IDL -Interface Definition Language.

image font -A set of symbols, each of which is described in a rectangular array of pels. Some of the pels in the array are set to produce the image of one of the symbols. Contrast with *outline font* .

implementation binding -See *private header file* .

implied length -The definition of a specific length for a data type. An example of this is the Data Access Manager's typeInteger data type, which has a defined length of 4 bytes.

implied metaclass -Subclassing the metaclass of a parent class without a separate IDL for the resultant metaclass.

inactive frame -A frame that does not have the selection focus.

inactive part -A part that has no active display frames.

incorporate -To merge the data from one part into the contents of another part so that the merged data retains no separate identity as a part. Contrast with e*mbed* .

indirect manipulation -Interaction with an object through choices and controls.

information device context -A logical description of a data destination other than the screen (for example, a printer or plotter), but where no output will occur. Its purpose is to satisfy queries. See also *device context* .

information panel -A defined panel type characterized by a body containing only protected information.

Information Presentation Facility (IPF) -A facility provided by the OS/2 operating system, by which application developers can produce online documentation and context-sensitive online help panels for their applications.

inheritance -The passing of class resources or attributes from a parent class downstream in the class hierarchy to a child class. The new class inherits all the data and methods of the parent class without having to redefine them.

in-place editing -User manipulation of data in an embedded part without leaving the context of the document in which the part is displayed (for example, without opening a new window for the part).

input focus -(1) The area of a window where user interaction is possible using an input device, such as a mouse or the keyboard. (2) The position in the *active window* where a user's normal interaction with the keyboard will appear.

input router -An internal OS/2 process that removes messages from the system queue.

input/output control -A device-specific command that requests a function of a device driver.

insertion location descriptor record -A record of type typeInsertionLoc that consists of two keyword-specified descriptor records. The first is an object specifier record, and the data for the second is a constant that specifies the insertion location in relation to the OSA event object described by the object specifier record.

inside-out activation -A mode of user interaction in which a mouse click anywhere in a document activates the smallest possible enclosing frame and performs the appropriate selection action on the content element at the click location. OpenDoc uses inside-out selection. Contrast with *outside-in activation* .

inside-out selection -A mode of user interaction in which a mouse click anywhere in a document activates the smallest possible enclosing frame and performs the appropriate selection action on the content element at the click location. OpenDoc uses inside-out selection. Contrast with *outside-in* s*election* .

installable file system (IFS) -A file system in which software is installed when the operating system is started.

instance -A single occurrence of an object class that has a particular behavior. See also *object* .

instantiate -(1) To make an instance of; to replicate. (2) In object-oriented programming, to represent a class abstraction with a concrete instance of the class.

instruction pointer -A pointer that provides addressability for a machine interface instruction in a program.

integer atom -An *atom* that represents a predefined system constant and carries no storage overhead. For example, names of window classes provided by Presentation Manager are expressed as integer atoms.

interactive graphics -Graphics that can be moved or manipulated by a user at a terminal.

interactive program -(1) A program that is running (active) and is ready to receive (or is receiving) input from a user. (2) A running program that can receive input from the keyboard or another input device. Contrast with *active program* and *noninteractive program* .

Also known as a *foreground program* .

interapplication communication (IAC) architecture -A standard and extensible mechanism for communication among applications, including the Open Scripting Architecture, the OSA Event Manager, and the Event Manager.

interchange file -A file containing data that can be sent from one application to another.

Interface Definition Language (IDL) -Language-neutral syntax created by IBM to describe the interface of classes that can be compiled by the SOM compiler.

internal transform -A transform that positions, scales, or otherwise transforms the image of a part drawn within a frame. Contrast with *external transform* .

internalize -For a part or other OpenDoc object, to transform its persistent form in a storage unit into an appropriate in-memory representation. Contrast with *externalize* . See also *read* .

interoperability -Access to an OpenDoc part or document from different platforms or with different software systems.

interpreter -A program that translates and executes each instruction of a high-level programming language before it translates and executes.

interprocess communication (IPC) -In the OS/2 operating system, the exchange of information between processes or threads through semaphores, pipes, queues, and shared memory.

interval timer -(1) A timer that provides program interruptions on a program-controlled basis. (2) An electronic counter that counts intervals of time under program control.

intrinsic content -The content elements native to a particular part, as opposed to other parts embedded in it. Contrast with *embedded content* .

invalid shape -The area of a frame, facet, or canvas that needs redrawing. Update events cause redrawing of the invalid area.

invalidate -To mark an area of a canvas (or facet, or frame) as in need of redrawing.

invariant -An aspect of the internal state of an object that must be maintained for the object to behave properly according to its design.

IOCtl -Input/output control.

IOPL -Input/output privilege level.

IOPL code segment -An IOPL executable section of programming code that enables an application to directly manipulate hardware interrupts and ports without replacing the device driver. See also *privilege level* .

IPC -Interprocess communication.

IPF -Information Presentation Facility.

IPF compiler -A text compiler that interpret tags in a source file and converts the information into the specified format.

IPF tag language -A markup language that provides the instructions for displaying online information.

ISO string -A null-terminated 7-bit ASCII string.

item -A data object that can be passed in a DDE transaction.

iterator -A class or object that provides sequential access to a collection of objects of another class. A part's embedded-frames iterator, for example, provides access to all of the part's embedded frames.

Glossary - J

journal -A special-purpose file that is used to record changes made in the system.

Glossary - K

Kanji -A graphic character set used in Japanese ideographic alphabets.

KBD$ -Character-device name reserved for the keyboard.

kernel -The part of an operating system that performs basic functions, such as allocating hardware resources.

kerning -The design of graphics characters so that their character boxes overlap. Used to space text proportionally.

key data -The data in an object specifier record that distinguishes one or more OSA event objects from other OSA event objects of the same object class in the same container. Key data is specified by a keyword-specified descriptor record with the keyword keyAEKeyData. The OSA Event Manager interprets key data according to the key form specified in the same object specifier record.

key form -The form taken by the key data in an object specifier record. The key form is specified by a keyword-specified descriptor record with the keyword keyAEKeyForm. The keyword-specified descriptor record contains a constant that determines how the OSA Event Manager and a target application use the key data to locate specific OSA event objects. For example, the key form constant formName indicates that the key data consists of a name, which should be compared to the names of OSA event objects in the container specified by the object specifier record.

keyboard accelerator -A keystroke that generates a command message for an application.

keyboard augmentation -A function that enables a user to press a keyboard key while pressing a mouse button.

keyboard focus -A temporary attribute of a window. The window that has a keyboard focus receives all keyboard input until the focus changes to a different window.

Keys help -In SAA Common User Access architecture, a help action that provides a listing of the application keys and their assigned functions.

keystroke focus -A designation of ownership of keystroke events. The part whose frame has the keystroke focus receives keystroke events. See also *selection focus* .

keystroke focus frame -The frame to which keystroke events are to be sent.

keyword -A four-character code that uniquely identifies a descriptor record inside another descriptor record. In OSA Event Manager functions, constants are typically used to represent the four-character codes.

keyword-specified descriptor record -A record of data type AEKeyDesc that consists of a keyword and a descriptor record. Keyword-specified descriptor records are used to describe the attributes and parameters of an OSA event.

kind -See *part kind* .

Glossary - L

label -In a graphics segment, an identifier of one or more elements that is used when editing the segment.

LAN -local area network.

language support procedure -A function provided by the Presentation Manager Interface for applications that do not, or cannot (as in the case of COBOL and FORTRAN programs), provide their own dialog or window procedures.

large icon view type -A view type in which a part is represented by a 32 by 32-pixel bitmap image. Other possible view types for displaying a part include small icon, thumbnail, and frame.

layout -The process of arranging frames and content elements in a document for drawing.

lazy drag -See *pickup and drop* .

lazy drag set -See *pickup set* .

lazy instantiation -The process of creating objects (such as embedded frames) in memory only when they are needed for display, such as

when the user scrolls them into view. Lazy instantiation can help minimize the memory requirements of your parts.

LDT -In the OS/2 operating system, Local Descriptor Table.

leaf part -See *noncontainer part* .

LIFO stack -A stack from which data is retrieved in last-in, first-out order.

linear address -A unique value that identifies the memory object.

link -(1) A persistent reference to a part or to a set of content elements of a part. (2) An OpenDoc object that represents a link destination.

link destination -The portion of a part's content area that represents the destination of a link.

link key -A number that identifies a specific transaction to access a link object or link-source object.

link manager -An OpenDoc object that coordinates cross-document links.

link source -The portion of a part's content area that represents the source of a link.

link specification -An object, placed on the clipboard or in a drag-and-drop object, from which the source part (the part that placed the data) can construct a link if necessary.

link status -The link-related state (in a link source, in a link destination, or not in a link) of a frame.

linked list -Synonym for *chained list* .

linked part -A part (or a portion of a part's content data) that appears to the user to be embedded in one part, but it is actually embedded in a different part. Linked data is not copied when the link's containing part is duplicated; a new link is created instead.

list box -In SAA Advanced Common User Access architecture, a control that contains scrollable choices from which a user can select one choice.

Note: In CUA architecture, this is a programmer term. The end user term is selection list.

list button -A button labeled with an underlined down-arrow that presents a list of valid objects or choices that can be selected for that field.

list panel -A defined panel type that displays a list of items from which users can select one or more choices and then specify one or more actions to work on those choices.

load -For a part editor, to transform the persistent form of a part in a draft into an appropriate in-memory representation, which can be a representation of the complete part or only a subset, depending on the current display requirements of the document. Contrast with *save* .

load-on-call -A function of a linkage editor that allows selected segments of the module to be disk resident while other segments are executing. Disk resident segments are loaded for execution and given control when any entry point that they contain is called.

load time -The point in time at which a program module is loaded into main storage for execution.

local area network (LAN) -(1) A computer network located on a user's premises within a limited geographical area. Communication within a local area network is not subject to external regulations; however, communication across the LAN boundary may be subject to some form of regulation. (T)

Note: A LAN does not use store and forward techniques. (2) A network in which a set of devices are connected to one another for communication and that can be connected to a larger network.

Local Descriptor Table (LDT) -Defines code and data segments specific to a single task.

lock -A serialization mechanism by means of which a resource is restricted for use by the holder of the lock.

logical descriptor record -A coerced AE record of type typeLogicalDescriptor that specifies a logical expression-that is, an expression that the OSA Event Manager evaluates to either TRUE or FALSE. The logical expression is constructed from a logical operator (one of the Boolean operators AND, OR, or NOT) and a list of logical terms to which the operator is applied. Each logical term in the list can be either another logical descriptor record or a comparison descriptor record.

logical storage device -A device that the user can map to a physical (actual) device.

LPT1, LPT2, LPT3 -Character-device names reserved for parallel printers 1 through 3.

Glossary - M

main storage unit -The storage unit that holds the contents property (kODPropContents) of a part. A part's main storage unit, plus possibly other auxiliary storage units referenced from it, holds all of a part's content.

main window -The window that is positioned relative to the *desktop window* .

manipulation button -The button on a pointing device a user presses to directly manipulate an object.

map -(1) A set of values having a defined correspondence with the quantities or values of another set. (I) (A) (2) To establish a set of values having a defined correspondence with the quantities or values of another set. (I)

mark-adjusting function -A marking callback function that unmarks objects previously marked by a call to an application's marking function.

mark count -The number of times the OSA Event Manager has called the marking function for the current mark token. Applications that support marking callback functions should associate the mark count with each OSA event object they mark.

mark token -A token returned by a mark token function. A mark token identifies the way an application marks OSA event objects during the current sessions while resolving a single test. A mark token does not identify a specific OSA event object; rather, it allows an application that supports marking callback functions to associate a group of objects with a marked set.

mark token function -A marking callback function that returns a mark token.

marker box -In computer graphics, the boundary that defines, in world coordinates, the horizontal and vertical space occupied by a single marker from a marker set.

marker symbol -A symbol centered on a point. Graphs and charts can use marker symbols to indicate the plotted points.

marking callback functions -Object callback functions that allow your application to use its own marking scheme rather than tokens when identifying large groups of OSA event objects. See also *mark-adjusting function* , *mark token function* , *object callback function* , and *object-marking function* .

marquee box -The rectangle that appears during a selection technique in which a user selects objects by drawing a box around them with a pointing device.

Master Help Index -In the OS/2 operating system, an alphabetic list of help topics related to using the operating system.

maximize -To enlarge a window to its largest possible size.

media window -The part of the physical device (display, printer, or plotter) on which a picture is presented.

member function -See *method* .

memory block -Part memory within a heap.

memory device context -A logical description of a data destination that is a memory bit map. See also *device context* .

memory management -A feature of the operating system for allocating, sharing, and freeing main storage.

memory object -Logical unit of memory requested by an application, which forms the granular unit of memory manipulation from the application viewpoint.

menu -In SAA Advanced Common User Access architecture, an extension of the menu bar that displays a list of choices available for a selected choice in the menu bar. After a user selects a choice in menu bar, the corresponding menu appears. Additional pop-up windows can appear from menu choices.

menu bar -In SAA Advanced Common User Access architecture, the area near the top of a window, below the title bar and above the rest of the window, that contains choices that provide access to other menus.

menu button -The button on a pointing device that a user presses to view a pop-up menu associated with an object.

message -(1) In the Presentation Manager, a packet of data used for communication between the Presentation Manager interface and Presentation Manager applications (2) In a user interface, information not requested by users but presented to users by the computer in response to a user action or internal process. See also s*emantic event* .

message block -A byte stream that an open application uses to send data to and receive data from another open application (which can be located on the same computer or across a network).

message box -(1) A dialog window predefined by the system and used as a simple interface for applications, without the necessity of creating dialog-template resources or dialog procedures. (2) In SAA Advanced Common User Access architecture, a type of window that shows messages to users. See also *dialog box, primary window,* s*econdary window* .

message filter -The means of selecting which messages from a specific window will be handled by the application.

message interface -An OpenDoc object that provides an interface to allow parts to send messages (semantic events) to other parts, either in the same document or in other documents.

message queue -A sequenced collection of messages to be read by the application.

message stream mode -A method of operation in which data is treated as a stream of messages. Contrast with *byte stream* .

metacharacter -See *global file-name character* .

metaclass -The conjunction of an object and its class information; that is, the information pertaining to the class as a whole, rather than to a single instance of the class. Each class is itself an object, which is an instance of the metaclass.

metafile -A file containing a series of attributes that set color, shape and size, usually of a picture or a drawing. Using a program that can interpret these attributes, a user can view the assembled image.

metafile device context -A logical description of a data destination that is a metafile, which is used for graphics interchange. See also *device context* .

metalanguage -A language used to specify another language. For example, data types can be described using a metalanguage so as to make the descriptions independent of any one computer language.

method -A function that manipulates the data of a particular class of objects.

method override -The replacement, by a child class, of the implementation of a method inherited from a parent and an ancestor class.

mickey -A unit of measurement for physical mouse motion whose value depends on the mouse device driver currently loaded.

micro presentation space -A graphics presentation space in which a restricted set of the GPI function calls is available.

minimize -To remove from the screen all windows associated with an application and replace them with an icon that represents the application.

mix -An attribute that determines how the foreground of a graphic primitive is combined with the existing color of graphics output. Also known as *foreground mix* . Contrast with *background mix* .

mixed character string -A string containing a mixture of one-byte and *Kanji* or Hangeul (two-byte) characters.

mnemonic -(1) A method of selecting an item on a pull-down by means of typing the highlighted letter in the menu item. (2) In SAA Advanced Common User Access architecture, usually a single character, within the text of a choice, identified by an underscore beneath the character. If all characters in a choice already serve as mnemonics for other choices, another character, placed in parentheses immediately following the choice, can be used. When a user types the mnemonic for a choice, the choice is either selected or the cursor is moved to that choice.

modal dialog box -In SAA Advanced Common User Access architecture, a type of movable window, fixed in size, that requires a user to enter information before continuing to work in the application window from which it was displayed. Contrast with *modeless dialog box* . Also known as a *serial dialog box* . Contrast with *parallel dialog box* .

Note: In CUA architecture, this is a programmer term. The end user term is pop-up window.

modal focus -A designation of ownership of the right to display modal dialog boxes. A part displaying a modal dialog must first acquire the modal focus, so that other parts cannot do the same until the first part is finished.

model space -See *graphics model space* .

modeless dialog box -In SAA Advanced Common User Access architecture, a type of movable window, fixed in size, that allows users to continue their dialog with the application without entering information in the dialog box. Also known as a *parallel dialog box* . Contrast with *modal dialog box* .

Note: In CUA architecture, this is a programmer term. The end user term is pop-up window.

module definition file -A file that describes the code segments within a load module. For example, it indicates whether a code segment is loadable before module execution begins (preload), or loadable only when referred to at run time (load-on-call).

monitor -A special use of a dispatch module, in which it is installed in order to be notified of events, but does not dispatch them.

monolithic application -See *conventional application* .

mouse -In SAA usage, a device that a user moves on a flat surface to position a pointer on the screen. It allows a user to select a choice o function to be performed or to perform operations on the screen, such as dragging or drawing lines from one position to another.

mouse region -An area (by default a size of 1 pixel square) within which the user can move the mouse pointer without triggering an event.

MOUSE$ -Character-device name reserved for a mouse.

multiple-choice selection -In SAA Basic Common User Access architecture, a type of field from which a user can select one or more choices or select none. See also *check box* . Contrast with *extended-choice selection* .

multiple-line entry field -In SAA Advanced Common User Access architecture, a control into which a user types more than one line of information. See also *single-line entry field* .

multitasking -The concurrent processing of applications or parts of applications. A running application and its data are protected from other concurrently running applications.

mutex semaphore -(Mutual exclusion semaphore). A semaphore that enables threads to serialize their access to resources. Only the thread that currently owns the mutex semaphore can gain access to the resource, thus preventing one thread from interrupting operations being performed by another.

muxwait semaphore -(Multiple wait semaphore). A semaphore that enables a thread to wait either for multiple event semaphores to be posted or for multiple mutex semaphores to be released. Alternatively, a muxwait semaphore can be set to enable a thread to wait for any ONE of the event or mutex semaphores in the muxwait semaphore's list to be posted or released.

Glossary - N

name resolver -An OpenDoc object that determines the proper recipient of a semantic event. The name resolver can resolve *object* specifiers, permitting semantic events to be sent to individual objects within a part.

name space -An object consisting of a set of text strings used to identify kinds of objects or classes of behavior, for registration purposes. For example, OpenDoc uses name spaces to identify part kinds and categories for binding.

name-space manager -An OpenDoc object that creates and deletes *name spaces* .

named pipe -A named buffer that provides client-to-server, server-to-client, or full duplex communication between unrelated processes. Contrast with *unnamed pipe* .

national language support (NLS) -The modification or conversion of a United States English product to conform to the requirements of another language or country. This can include the enabling or retrofitting of a product and the translation of nomenclature, MRI, or documentation of a product.

nested list -A list that is contained within another list.

NLS -national language support.

non-8.3 file-name format -A file-naming convention in which file names can consist of up to 255 characters. See also *8.3 file-name format* .

non-exclusive focus -A focus that can be owned by more than one frame at a time. OpenDoc supports the use of nonexclusive foci. Contrast with *exclusive focus* .

noncontainer part -A part that cannot itself contain embedded parts. A noncontainer part can never be a *containing part* . Contrast with *container part* .

noncritical extended attribute -An extended attribute that is not necessary for the function of an application.

nondestructive read -Reading that does not erase the data in the source location. (T)

nonembedding part -see *noncontainer part* . Contrast with *embedding part* .

noninteractive program -A running program that cannot receive input from the keyboard or other input device. Contrast with *active program* and *interactive program* .

nonpersistent frame -A frame that exists as an object in memory, but has no storage unit and is not stored persistently.

nonretained graphics -Graphic primitives that are not remembered by the Presentation Manager interface when they have been drawn. Contrast with *retained graphics* .

null character (NUL) -(1) Character-device name reserved for a nonexistent (dummy) device. (2) A control character that is used to accomplish media-fill or time-fill and that may be inserted into or removed from a sequence of characters without affecting the meaning of the sequence; however, the control of equipment or the format may be affected by this character. (I) (A)

null descriptor record -A descriptor record whose descriptor type is typeNull and whose data handle is NIL.

null-terminated string -A string of (n+1) characters where the (n+1)th character is the 'null' character (0x00) Also known as 'zero-terminated' string and 'ASCIIZ' string.

Glossary - O

object - A programming entity, existing in memory at run time, that is an individual instantiation of a particular *class* .

object accessor -A function called by the name resolver to resolve semantic-event object specifiers.

object accessor dispatch table -A table in shared memory that the OSA Event Manager uses to map descriptions of objects in an object specifier record to object accessor functions that can locate those objects.

object accessor function -An application-defined function that locates an OSA event object of a specified object class in a container identified by a token of a specified descriptor type.

object callback -A function called by the name resolver to allow your part to provide extra information needed for semantic-event object resolution.

object callback function -An application-defined function used by the OSA Event Manager to resolve object specifier records. See also *error callback function* , *marking callback function*s , *object-comparison function* , *object-counting function* , and *token disposal function* .

object class -A category for OSA event objects that share specific characteristics listed in an object class definition in the *OSA Event Registry: Standard Suite*s Among these characteristics are properties, element classes, and OSA events that can specify objects of that class. An object class is specified in an object specifier record by a keyword-specified descriptor record with the keyword keyAEDesiredClass whose data handle refers to either a constant or an object class ID.

object class ID -A four-character code, which can also be represented by a constant, that identifies an object class for an OSA event object. The object class ID for a primitive object class is the same as the four-character value of its descriptor type.

object class inheritance hierarchy -The hierarchy of subclasses and superclasses that determines which properties, elements, and OSA events object classes inherit from other object classes.

object-comparison function -An object callback function that compares an element to either another element or to a descriptor record and returns either TRUE or FALSE.

object-counting function -An object callback function that counts the number of elements of a specified class in a specified container, so that the OSA Event Manager can determine how many elements it must examine to find the element or elements that pass a test.

Object Interface Definition Language (OIDL) -Specification language used in SOM Version 1 for defining classes. Replaced by Interface Definition Language (IDL).

Object Linking and Embedding (OLE) -An application protocol developed by Microsoft Corporation that allows objects created by one application to be linked to or embedded in objects created by another application.

Object Management Group (OMG) -An industry consortium that promulgates standards for object programming.

object-marking function -An object callback function called repeatedly by the OSA Event Manager to mark specific OSA event objects. See also *marking callback function*s .

object model -A feature of OSA events that allows a part to define a hierarchical arrangement of content objects to represent the elements of the part's content.

Object REXX component -The scripting component that implements the Object REXX scripting language. See also *scripting component* .

Object REXX scripting language -The standard user scripting language defined by IBM. The Object REXX scripting language is implemented by the Object REXX scripting component.

object specifier -A designation of a content object within a part, used to determine the target of a semantic event. Object specifiers can be names ("blue rectangle") or logical designations ("word 1 of line 2 of embedded frame 3").

object specifier record -A description of one or more OSA event objects based on the OSA Event Manager and the classification system defined in the *OSA Event Registry: Standard Suite*s . An object specifier record consists of a descriptor record of descriptor type typeObjectSpecifier that comprises four keyword-specified descriptor records: the object class ID, the container for the OSA event object (which is usually another OSA event object, specified by another object specifier record), the key form, and the key data.

object window -A window that does not have a parent but which might have child windows. An object window cannot be presented on a device.

OIDL -Object Interface Definition Language.

OLE -See *Object Linking and Embedding* .

OLE interoperability -A technology that enables seamless interoperability between OpenDoc and Microsoft Corporation's Object Linking and Embedding (OLE) technology for interapplication communication. It allows OLE objects to function automatically as parts in OpenDoc documents, and OpenDoc parts to function automatically as OLE objects in OLE containers.

open -To start working with a file, directory, or other object.

Open Application event -An OSA event that asks an application to perform the tasks-such as displaying untitled windows-associated with opening itself; one of the four required OSA events.

Open Documents event -An OSA event that asks an application to open one or more documents specified in a list; one of the four required OSA events.

Open Linking and Embedding of Objects (OLEO) -A technology that enables seamless interoperability between OpenDoc and Microsoft Corporation's Object Linking and Embedding (OLE) technology for interapplication communication. It allows OLE objects to function automatically as parts in OpenDoc documents, and OpenDoc parts to function automatically as OLE objects in OLE containers.

Open Scripting Architecture (OSA) -A mechanism based on the OSA Event Manager and the *OSA Event Registry: Standard Suite*s that allows users to control multiple applications by means of scripts. The scripts can be written in any scripting language that supports the OSA.

OpenDoc -A multiplatform technology, implemented as a set of shared libraries, that uses component software to facilitate the construction and sharing of compound documents.

OpenDoc Development Framework (ODF) -A part-editor framework that facilitates creation of OpenDoc parts.

optional parameter -A supplemental parameter in an OSA event used to specify data that the server application can use in addition to the data specified in the direct parameter. Source applications list the keywords for parameters that they consider optional in the attribute identified by the keyOptionalKeywordAttr keyword. Target applications use this attribute to identify any parameters that they are required to understand. If a parameter's keyword is not listed in this attribute, the target application must understand that parameter to handle the event successfully. See also *OSA event attribute* , *OSA event parameter* .

ordered list -Vertical arrangements of items, with each item in the list preceded by a number or letter.

OSA event -An OSA event consists of attributes (including the event class and event ID, which identify the event and its task) and, usually, parameters (which contain data used by the target application for the event). See also *OSA event attribute* , *OSA event parameter* .

OSA event array -An array in a descriptor list. The data for an OSA event array is specified by an array data record, which is defined by the data type AEArrayData.

OSA event attribute -A keyword-specified descriptor record that identifies the event class, event ID, target application, or some other characteristic of an OSA event. Taken together, the attributes of an OSA event identify the event and denote the task to be performed on the data specified in the OSA event's parameters. Unlike OSA event parameters (which contain data used only by the target application of the OSA event), OSA event attributes contain information that can be used by both the OSA Event Manager and the target application. See also *OSA event parameter* .

OSA event dispatch table -A table in shared memory that the OSA Event Manager uses to map OSA events to the appropriate OSA event handlers.

OSA event handler -An application-defined function that extracts pertinent data from an OSA event, performs the action requested by the OSA event, and returns a result.

OSA Event Manager -The collection of routines that allows client applications to send OSA events to server applications for the purpose of requesting services or information.

OSA event object -A distinct item in a target application or any of its documents that can be specified by an object specifier record in an OSA event sent by a source application. OSA event objects can be anything that an application can locate on the basis of such a description, including items that a user can differentiate and manipulate while using an application, such as words, paragraphs, shapes, windows, or style formats. See also *object specifier record* .

OSA event object class -See *object class* .

OSA event parameter -A keyword-specified descriptor record containing data that the target application for an OSA event uses. Unlike OSA event attributes (which contain information that can be used by both the OSA Event Manager and the target application), OSA event parameters contain data used only by the target application of the OSA event. See also *OSA event attribute* , *direct parameter* , *optional parameter* , *required parameter* .

OSA event record -A descriptor record of data type OSAEvent that contains a list of keyword-specified descriptor records. These descriptor records describe-at least-the attributes necessary for an OSA event; they may also describe parameters for the OSA event. OSA Event Manager functions are used to add parameters to an OSA event record.

OSA event user terminology resources -Two resources with identical formats used by server applications to specify the OSA events and corresponding user terminology that the applications support. The 'aeut' resource, which is provided by scripting components, contains terminology information for all the standard suites of OSA events defined in the *OSA Event Registry: Standard Suite*s . An 'aete' resource must be provided by every scriptable application; it describes which of the standard suites listed in the 'aeut' resource the application supports and provides additional terminology information for extensions to the standard suites and custom OSA events supported by the application. See also *scripting component* .

outline font -A set of symbols, each of which is created as a series of lines and curves. Synonymous with *vector font* . Contrast with *image font* .

output area -An area of storage reserved for output. (A)

outside-in activation -A mode of user interaction in which a mouse click anywhere in a document activates the largest possible enclosing frame that is not already active. Contrast with *inside-out activation* .

outside-in selection -A mode of user interaction in which a mouse click anywhere in a document activates the largest possible enclosing frame that is not already active. Contrast with *inside-out selection* .

overlaid frame -An embedded frame that floats above the content (including other embedded frames) of its containing part, and thus need not engage in frame negotiation with the containing part.

override -To replace a method belonging to a superclass with a method of the same name in a subclass, in order to modify its behavior.

owner -For a canvas, the part that created the canvas and attached it to a facet. The owner is responsible for transferring the results of drawing on the canvas to its parent canvas.

owner window -A window into which specific events that occur in another (owned) window are reported.

ownership -The determination of how windows communicate using messages.

owning process -The process that owns the resources that might be shared with other processes.

Glossary - P

page -(1) A 4KB segment of contiguous physical memory. (2) A defined unit of space on a storage medium.

page viewport -A boundary in device coordinates that defines the area of the output device in which graphics are to be displayed. The presentation-page contents are transformed automatically to the page viewport in device space.

paint -(1) The action of drawing or redrawing the contents of a window. (2) In computer graphics, to shade an area of a display image; for example, with crosshatching or color.

panel -In SAA Basic Common User Access architecture, a particular arrangement of information that is presented in a window or pop-up. If some of the information is not visible, a user can scroll through the information.

panel area -An area within a panel that contains related information. The three major Common User Access-defined panel areas are the action bar, the function key area, and the panel body.

panel area separator -In SAA Basic Common User Access architecture, a solid, dashed, or blank line that provides a visual distinction between two adjacent areas of a panel.

panel body -The portion of a panel not occupied by the action bar, function key area, title or scroll bars. The panel body can contain protected information, selection fields, and entry fields. The layout and content of the panel body determine the panel type.

panel body area -See *client area* .

panel definition -A description of the contents and characteristics of a panel. A panel definition is the application developer's mechanism for predefining the format to be presented to users in a window.

panel ID -In SAA Basic Common User Access architecture, a panel identifier, located in the upper-left corner of a panel. A user can choose whether to display the panel ID.

panel title -In SAA Basic Common User Access architecture, a particular arrangement of information that is presented in a window or pop-up. If some of the information is not visible, a user can scroll through the information.

paper size -The size of paper, defined in either standard U.S. or European names (for example, A, B, A4), and measured in inches or millimeters respectively.

parallel dialog box -See *modeless dialog box* .

parameter list -A list of values that provides a means of associating addressability of data defined in a called program with data in the calling program. It contains parameter names and the order in which they are to be associated in the calling and called program.

parent canvas -The canvas closest above a canvas in the facet hierarchy. If, for example, there is a single offscreen canvas attached to an embedded facet in a window, the window canvas (attached to the root facet) is the parent of the offscreen canvas.

parent class -See *superclass* .

parent process -In the OS/2 operating system, a process that creates other processes. Contrast with *child process* .

parent window -In the OS/2 operating system, a window that creates a child window. The child window is drawn within the parent window. If the parent window is moved, resized, or destroyed, the child window also will be moved, resized, or destroyed. However, the child window can be moved and resized independently from the parent window, within the boundaries of the parent window. Contrast with *child window* .

part -A portion of a compound document; it consists of document content, plus-at run time-a part editor that manipulates that content. The content is data of a given structure or type, such as text, graphics, or video; the code is a part editor. In programming terms, a part is an object, an instantiation of a subclass of the class ODpart. To a user, a part is a single set of information displayed and manipulated in one or more frames or windows. Synonymous with *document part* .

part category -A general classification of the format of data handled by a part editor. Categories are broad classes of data format, meaningful to end-users, such as "text", "graphics", or "table". Contrast with *part kind* .

part container -See *container part* .

part content -The portion of a part that describes its data. In programming terms, the part content is represented by the instance variables of the part object; it is the state of the part and is the portion of it that is stored persistently. To the user, there is no distinction between part and part content; the user considers both the part content alone, and the content plus its part editor, as a part. Contrast with *part* editorand *part* . See also *intrinsic content* and *embedded content* .

part editor -An application component that can display and change the data of a part. It is the executable code that provides the behavior for the part. Contrast with *part content, part viewer* .

part ID -An identifier that uniquely names a part within the context of a document. This ID represents a storage unit ID within a particular draft of a document.

part info -(1) Part-specific data, of any type or size, used by a part editor to identify what should be displayed in a particular frame or facet and how it should be displayed. (2) Information about a given part that can be seen by the user and is displayed in the Part Info dialog box.

part kind -A specific classification of the format of data handled by a part editor. A kind specifies the specific data format handled by, and possibly native to, a part editor. Kinds are meaningful to end-users and have designations such as such as "MyEditor 2.0" or "MyEditor 1.0". Contrast with *part category* .

part property -A user-accessible characteristic of a part or a portion of its content. The user can modify some properties, such as the name of a part; the user cannot modify some other properties, such as its part category. See also *property* .

part registry -The mechanism by which the document shell maps parts to part editors according to their part kind.

part table -A list of all the parts contained within a document and a list of associated data.

part viewer -An application component that can display, but not change, the data of a part. Contrast with *part editor* .

part window -A window that displays an embedded part by itself, for easier viewing or editing. Any part that is embedded in another part can be opened up into its own part window. The part window is separate from and has a slightly different appearance than the *document window* that displays the entire document in which the part is embedded.

part-wrapper object -A private OpenDoc object that is used to reference a part.

partition -(1) A fixed-size division of storage. (2) On an IBM personal computer fixed disk, one of four possible storage areas of variable size; one may be accessed by DOS, and each of the others may be assigned to another operating system.

Paste -A choice in the **Edit** pull-down that a user selects to move the contents of the clipboard into a preselected location. See also *Copy* and *Cut* .

Paste As -A choice in the **Edit** pull-down that a user selects to move the contents of the clipboard into a preselected location by means of a dialog box allowing the user to specify the format of the data. See also *Copy* and *Cut* .

path -The route used to locate files; the storage location of a file. A fully qualified path lists the drive identifier, directory name, subdirectory name (if any), and file name with the associated extension.

PDD -Physical device driver.

peeking -An action taken by any thread in the process that owns the queue to examine queue elements without removing them.

pel -(1) The smallest area of a display screen capable of being addressed and switched between visible and invisible states. Synonym for *display point* , *pixel* , and *picture element* . (2) Picture element.

persistence -The quality of an entity such as a part, link, or object, that allows it to span separate document launches and transport to different computers. For example, a part unloaded to persistent storage is typically written to a hard disk.

persistent object -An object whose instance data and state are preserved between system shutdown and system startup.

persistent reference -A number, stored somewhere within a storage unit, that refers to another storage unit in the same document. Persistent references permit complex runtime object relationships to be stored externally, and later reconstructed.

physical device driver (PDD) -A system interface that handles hardware interrupts and supports a set of input and output functions.

pick -To select part of a displayed object using the pointer.

pickup -To add an object or set of objects to the pickup set.

pickup and drop -A drag operation that does not require the direct manipulation button to be pressed for the duration of the drag.

pickup set -The set of objects that have been picked up as part of a pickup and drop operation.

picture chain -See s*egment chain* .

picture element -(1) Synonym for *pel* . (2) In computer graphics, the smallest element of a display surface that can be independently assigned color and intensity. (T) . (3) The area of the finest detail that can be reproduced effectively on the recording medium.

PID -Process identification.

pipe -(1) A named or unnamed buffer used to pass data between processes. A process reads from or writes to a pipe as if the pipe were a standard-input or standard-output file. See also *named pipe* and *unnamed pip*e . (2) To direct data so that the output from one process becomes the input to another process. The standard output of one command can be connected to the standard input of another with the pipe operator (|).

pixel -(1) Synonym for *pel* . (2) Picture element.

platform -The operating system environment in which a program runs. For example, OpenDoc is implemented on the OS/2, Macintosh, and Windows platforms.

platform-normal coordinates -The native coordinate system for a particular platform. OpenDoc performs all layout and drawing in platform-normal coordinates; to convert from another coordinate system to platform- normal coordinates requires application of a *bia*s *transform* .

plotter -An output unit that directly produces a hardcopy record of data on a removable medium, in the form of a two-dimensional graphic representation. (T)

PM -Presentation Manager.

pointer -(1) The symbol displayed on the screen that is moved by a pointing device, such as a *mou*se . The pointer is used to point at items that users can select. Contrast with *cursor* . (2) A data element that indicates the location of another data element. (T)

POINTER$ -Character-device name reserved for a pointer device (mouse screen support).

pointing device -In SAA Advanced Common User Access architecture, an instrument, such as a mouse, trackball, or joystick, used to move a pointer on the screen.

pointings -Pairs of x-y coordinates produced by an operator defining positions on a screen with a pointing device, such as a *mou*se .

polyfillet -A curve based on a sequence of lines. The curve is tangential to the end points of the first and last lines, and tangential also to the midpoints of all other lines. See also *fillet* .

polygon -One or more closed figures that can be drawn filled, outlined, or filled and outlined.

polyline -A sequence of adjoining lines.

polymorphism -The ability to have different implementations of the same method for two or more classes of objects.

pop -To retrieve an item from a last-in-first-out stack of items. Contrast with *push* .

pop-up menu -A menu that lists the actions that a user can perform on an object. The contents of the pop-up menu can vary depending on the context, or state, of the object.

pop-up window -(1) A window that appears on top of another window in a dialog. Each pop-up window must be completed before returning to the underlying window. (2) In SAA Advanced Common User Access architecture, a movable window, fixed in size, in which a user provides information required by an application so that it can continue to process a user request.

port -(1) A connection between the CPU and main memory or a device (such as a terminal) for transferring data. (2) A socket on the back panel of a computer where you plug in a cable for connection to a network or a peripheral device.

port name -A unique identifier for a particular application within a computer. The port name contains a name string, a type string, and a script code. An application can specify any number of port names for a single port so long as each name is unique. See also *port* .

position code -A parameter (to a storage unit's Focus method) with which you specify the desired property or value to access.

preferences -The mechanism through which the user assigns a part editor to a given part kind.

preferred editor -The part editor that last edited a part, or for whom the part's data was just translated. If a part's preferred editor is not present, OpenDoc attempts to bind the part to the user's *default editor for kind* or *default editor for category* .

preferred kind -The part kind that a part specifies as its highest-fidelity, preferred format for editing. It is the part kind stored as the first value in the contents property of the part's storage unit, unless the storage unit also contains a property of type kODPropPreferredKind specifying another value as the preferred kind.

presentation -A particular style of display for a part's content-for example, an outline or expanded style for text, or a wire-frame or solid style for graphic objects. A part can have multiple presentations, each with its own rendering, layout, and user-interface behavior. Contrast with *view type* .

presentation drivers -Special purpose I/O routines that handle field device-independent I/O requests from the PM and its applications.

Presentation Manager (PM) -The interface of the OS/2 operating system that presents, in windows a graphics-based interface to applications and files installed and running under the OS/2 operating system.

presentation page -The coordinate space in which a picture is assembled for display.

presentation space (PS) -(1) Contains the device-independent definition of a picture. (2) The display space on a display device.

primary window -In SAA Common User Access architecture, the window in which the main interaction between the user and the application takes place. In a multiprogramming environment, each application starts in its own primary window. The primary window remains for the duration of the application, although the panel displayed will change as the user's dialog moves forward. See also s*econdary window* .

primitive -In computer graphics, one of several simple functions for drawing on the screen, including, for example, the rectangle, line, ellipse, polygon, and so on.

primitive attribute -A specifiable characteristic of a graphic primitive. See *graphics attribute*s .

primitive object class -An object class defined in the *OSA Event Registry: Standard Suite*s for OSA event objects that contain a single value; for example, the cBoolean, cLongInteger, and cAlias object classes are all primitive object classes. An OSA event object that belongs to a primitive object class has no properties and contains only one element of the value of the data.

Print Documents event -An OSA event that requests that an application print a list of documents; one of the four required OSA events.

print job -The result of sending a document or picture to be printed.

private header file -A SOM-generated file containing macros that provide access to instance variables and invoke superclass methods of a SOM class.

privilege level -A protection level imposed by the hardware architecture of the IBM personal computer. There are four privilege levels (number 0 through 3). Only certain types of programs are allowed to execute at each privilege level. See also *IOPL code* s*egment* .

procedure call -In programming languages, a language construct for invoking execution of a procedure.

process -An instance of an executing application and the resources it is using.

program -A sequence of instructions that a computer can interpret and execute.

program details -Information about a program that is specified in the *Program Manager* window and is used when the program is started.

program group -In the Presentation Manager, several programs that can be acted upon as a single entity.

program name -The full file specification of a program. Contrast with *program title* .

program title -The name of a program as it is listed in the *Program Manager* window. Contrast with *program name* .

promise -A specification of data to be transferred at a future time. If a data transfer involves a very large amount of data, the source part can choose to put out a promise instead of actually writing the data to a storage unit.

prompt -A displayed symbol or message that requests input from the user or gives operational information; for example, on the display screen of an IBM personal computer, the DOS A> prompt. The user must respond to the prompt in order to proceed.

Properties notebook -A control window that is used to display the properties for a part and to enable the user to change them.

property -(1) In the OpenDoc storage subsystem, a component of a storage unit. A property defines a kind of information (such as "name" or "contents") and contains one or more data streams, called *value*s , that consist of information of that kind. Properties in a stored part are accessible without the assistance of a part editor. Contrast with *content property* and *Info property* . See also *display property, user property,* and *OSA event property* . (2) An OSA event object that defines some characteristic of another OSA event object, such as its font or point size, that can be uniquely identified by a constant. The definition of each object class in the *OSA Event Registry: Standard Suite*s lists the constants and class IDs for properties of OSA event objects belonging to that object class. For example, the constants pName and pBounds identify the name and boundary properties of OSA event objects that belong to the object class cWindow. The pName property of a specific window is defined by an OSA event object of object class cProperty, such as the word "MyWindow," which defines the name of the window. An OSA event object can contain only one of each of its properties, whereas it can contain many elements of the same element class. See also *OSA event object* , *container* , e*lement class*es .

property ID -A four-character code, which can also be represented by a constant, that identifies an OSA property.

protect mode -A method of program operation that limits or prevents access to certain instructions or areas of storage. Contrast with *real mode* .

protocol -(1) A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. (I) (2) The programming interface through which a specific task or set of related tasks is performed. The drag-and-drop protocol, for example, is the set of calls that a part editor makes (and responds to) in order to support the dragging of items into or out of its content.

proxy -A special type of content element in a containing part, a proxy is the site of an embedded part. A proxy holds a frame that has a reference to an embedded part or linked part.

proxy content -Data, associated with a single embedded frame written to the Clipboard (or drag-and-drop object or link-source object), that the frame's original containing part wanted associated with the frame, such as a drop shadow or other visual adornment. Proxy content is absent if intrinsic content as well as an embedded frame was written.

pseudocode -An artificial language used to describe computer program algorithms without using the syntax of any particular programming language. (A)

public header file -A SOM-generated file containing the client interface of a SOM class.

pull-down -(1) An *action bar* extension that displays a list of choices available for a selected action bar choice. After users select an action bar choice, the pull-down appears with the list of choices. Additional *pop-up windows* may appear from pull-down choices to further extend the actions available to users. (2) In SAA Common User Access architecture, pertaining to a choice in an action bar pull-down.

purge -To free noncritical memory, usually by writing or releasing cached data. In low-memory situations, OpenDoc can ask a part editor or other objects to purge memory.

push -To add an item to a last-in-first-out stack of items. Contrast with *pop* .

push button -In SAA Advanced Common User Access architecture, a rectangle with text inside. Push buttons are used in windows for actions that occur immediately when the push button is selected.

putback -To remove an object or set of objects from the lazy drag set. This has the effect of undoing the pickup operation for those objects

putdown -To drop the objects in the lazy drag set on the target object.

Glossary - Q

queue -(1) A linked list of elements waiting to be processed in FIFO order. For example, a queue may be a list of print jobs waiting to be printed. (2) A line or list of items waiting to be processed; for example, work to be performed or messages to be displayed.

queued device context -A logical description of a data destination (for example, a printer or plotter) where the output is to go through the spooler. See also *device context* .

Quit Application event -An OSA event that requests that an application perform the tasks-such as releasing memory, asking the user to save documents, and so on-associated with quitting; one of the four required OSA events. The Finder sends this event to an application immediately after sending it a Print Documents event or if the user chooses Restart or Shut Down from the Finder's Special menu.

Glossary - R

radio button -(1) A control window, shaped like a round button on the screen, that can be in a checked or unchecked state. It is used to select a single item from a list. Contrast with *check box* . (2) In SAA Advanced Common User Access architecture, a circle with text beside it. Radio buttons are combined to show a user a fixed set of choices from which only one can be selected. The circle is partially filled when a choice is selected.

range descriptor record -A coerced AE record of type typeRangeDescriptor that identifies two OSA event objects marking the beginning and end of a range of elements. See also *boundary objects* .

RAS -Reliability, availability, and serviceability.

raster -(1) In computer graphics, a predetermined pattern of lines that provides uniform coverage of a display space. (T) (2) The coordinate grid that divides the display area of a display device. (A)

read-only file -A file that can be read from but not written to.

real mode -A method of program operation that does not limit or prevent access to any instructions or areas of storage. The operating system loads the entire program into storage and gives the program access to all system resources. Contrast with *protect mode* .

realize -To cause the system to ensure, wherever possible, that the physical color table of a device is set to the closest possible match in the logical color table.

recordable -A level of scripting support of a part. A recordable part allows the user to automatically convert user actions into scripts attached to the part. Contrast with s*criptable, customizable, tinkerable* .

recordable application -An application that uses OSA events to report user actions to the OSA Event Manager for recording purposes. When a user turns on recording (for example, by pressing the Record button in the Script Editor application), a scripting component translates the OSA events generated by the user's subsequent actions into statements in a scripting language and records them in a compiled script. See also s*criptable application* .

recordable event -Any OSA event that any recordable application sends to itself while recording is turned on for the local computer, with the exception of events that are sent with the kAEDontRecord flag set in the sendMode parameter of the AESend function.

recording process -Any process (for example, a script editor) that can turn OSA event recording on and off and receive and record recordable OSA events.

recursive routine -A routine that can call itself, or be called by another routine that was called by the recursive routine.

reentrant -The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

reference -A pointer to (or other representation of) an object, used to gain access to the object when needed.

reference count -The number of references to an object. Objects that are reference-counted, such as windows and parts, cannot be deleted from memory unless their reference counts are zero.

reference counted object -An object that maintains a reference count. All classes descended from ODRefCntObject are reference-counted.

reference phrase -(1) A word or phrase that is emphasized in a device-dependent manner to inform the user that additional information for the word or phrase is available. (2) In hypertext, text that is highlighted and preceded by a single-character input field used to signify the existence of a hypertext link.

reference phrase help -In SAA Common User Access architecture, highlighted words or phrases within help information that a user selects to get additional information.

refresh -To update a window, with changed information, to its current status.

region -A clipping boundary in device space.

register -A part of internal storage having a specified storage capacity and usually intended for a specific purpose. (T)

registry -A dictionary that lists executable code modules and associated data by which they can be selected. Examples of a registry are *part registry* and s*cripting component registry* .

release -To delete a reference to an object. For a reference-counted object, releasing it decrements its reference count.

remote file system -A file-system driver that gains access to a remote system without a block device driver.

remove -To delete an object (such as a frame) permanently from its draft, as well as from memory. Contrast with *close* .

required OSA event -One of the four OSA events in the Required suite: Open Documents, Open Application, Print Documents, or Quit Application.

required parameter -An OSA event parameter that must be included in an OSA event. For example, a list of documents to open is a required parameter for the Open Documents event. Direct parameters are often required, and other additional parameters may be required. Optional parameters are never required.

resolve -To locate the OSA event object described by an object specifier record.

resource -The means of providing extra information used in the definition of a window. A resource can contain definitions of fonts, templates, accelerators, and mnemonics; the definitions are held in a resource file.

resource file -A file containing information used in the definition of a window. Definitions can be of fonts, templates, accelerators, and mnemonics.

restore -To return a window to its original size or position following a sizing or moving action.

retained graphics -Graphic primitives that are remembered by the Presentation Manager interface after they have been drawn. Contrast with *nonretained graphics* .

return code -(1) A value returned to a program to indicate the results of an operation requested by that program. (2) A code used to influence the execution of succeeding instructions.(A)

reverse video -(1) A form of highlighting a character, field, or cursor by reversing the color of the character, field, or cursor with its background; for example, changing a red character on a black background to a black character on a red background. (2) In SAA Basic Common User Access architecture, a screen emphasis feature that interchanges the foreground and background colors of an item.

revert -To return a draft to the state it had just after its last save.

REXX Language -Restructured Extended Executor. A procedural language that provides batch language functions along with structured programming constructs such as loops; conditional testing and subroutines.

RGB -(1) Color coding in which the brightness of the additive primary colors of light, red, green, and blue, are specified as three distinct values of white light. (2) Pertaining to a color display that accepts signals representing red, green, and blue.

roman -Relating to a type style with upright characters.

root facet -The facet that displays the root frame in a document window.

root frame -The frame in which the root part of a document is displayed. The root frame shape is the same as the content area of the document window.

root part -The part that forms the base of a document and establishes its basic editing, embedding, and printing behavior. A document has only one root part, which can contain content elements and perhaps other, embedded parts. Any part can be a root part.

root segment -In a hierarchical database, the highest segment in the tree structure.

root storage unit -See *content storage unit* .

root window -See *document window* .

round-robin scheduling -A process that allows each thread to run for a specified amount of time.

run time -(1) Any instant at which the execution of a particular computer program takes place. (T) (2) The amount of time needed for the execution of a particular computer program. (T) (3) The time during which an instruction in an instruction register is decoded and performed. Synonym for *execution time* .

Glossary - S

SAA -Systems Application Architecture.

save -To write all the data of all parts of a document (draft) to persistent storage.

SBCS -Single-byte character set.

scheduler -A computer program designed to perform functions such as scheduling, initiation, and termination of jobs.

scope -The range of a cloning operation, limiting which objects are to be copied. Scope is expressed in terms of a frame object or its storage unit.

screen -In SAA Basic Common User Access architecture, the physical surface of a display device upon which information is shown to a user.

screen device context -A logical description of a data destination that is a particular window on the screen. See also *device context* .

SCREEN$ -Character-device name reserved for the display screen.

script -A sequence of written instructions that, when executed by a script interpreter, are converted to semantic events that manipulate parts.

script application component -A component registered with the Component Manager at system startup. When a user opens a script application, the script application component loads the script and passes the resulting script ID to the appropriate scripting component for execution.

script data -A compiled script, script value, script context, or any other representation of a script in memory used internally by a scripting component. See also *compiled script* and *script value* .

script editor -An application that allows users to record, edit, save, and execute scripts; for example, the Script Editor application.

script file -A file in which a script is stored. A script file can be a compiled script file, a script application file, or a script text file.

script ID -A data structure of type OSAID-that is, a long integer-used by scripting components to keep track of script data.

script text file -Uncompiled statements in a scripting language saved by a script editor as a text file. A user must open a script text file in a script editor and successfully compile it before it will execute. See also *script editor* .

script value -An integer, a string, a Boolean value, a constant, or any other fixed data that a scripting component returns or uses in the course of executing a script.

scriptable -A level of scripting support of a part. A scriptable part is able to accept semantic events for its publicly published content objects and operations. Contrast with *customizable* , *tinkerable* and *recordable*. .

scriptable application -An application that can respond as a server application to OSA events sent to it by scripting components. To be scriptable, an application must respond to the appropriate standard OSA events, and it must provide an 'aete' resource that describes the nature of that support. See also *OSA event user terminology resources* .

scripting -Writing and executing scripts to control the behavior of multiple applications.

scripting component -A component that responds appropriately to calls made to the standard scripting component routines. Most scripting components implement scripting languages; for example, the Object REXX component implements the Object REXX scripting language.

scroll bar -In SAA Advanced Common User Access architecture, a part of a window, associated with a scrollable area, that a user interacts with to see information that is not currently allows visible.

scrollable entry field -An entry field larger than the visible field.

scrollable selection field -A selection field that contains more choices than are visible.

scrolling -Moving a display image vertically or horizontally in a manner such that new data appears at one edge, as existing data disappears at the opposite edge.

secondary window -A window that contains information that is dependent on information in a primary window and is used to supplement the interaction in the primary window.

sector -On disk or diskette storage, an addressable subdivision of a track used to record one block of a program or data.

segment -See *graphics segment* .

segment attributes -Attributes that apply to the segment as an entity, as opposed to the individual primitives within the segment. For example, the visibility or detectability of a segment.

segment chain -All segments in a graphics presentation space that are defined with the 'chained' attribute. Synonym for *picture chain* .

segment priority -The order in which segments are drawn.

segment store -An area in a normal graphics presentation space where retained graphics segments are stored.

select -(1) To mark or choose an item. Note that *select* means to mark or type in a choice on the screen; *enter* means to send all selected choices to the computer for processing. (2) In OpenDoc, to designate as the focus of subsequent editing operations. If the user selects an embedded part, that part's frame border takes on an appearance that designates it as selected. The embedded part's container is activated.

select button -The button on a pointing device, such as a mouse, that is pressed to select a menu choice. Also known as button 1.

selection cursor -In SAA Advanced Common User Access architecture, a visual indication that a user has selected a choice. It is represented by outlining the choice with a dotted box. See also *text cursor* .

selection field -(1) In SAA Advanced Common User Access architecture, a set of related choices. See also *entry field* . (2) In SAA Basic Common User Access architecture, an area of a panel that cannot be scrolled and contains a fixed number of choices.

selection focus -The location of editing activity. The part whose frame has the selection focus is the active part, and has the selection or insertion point. See also *keystroke focus* .

semantic event -A message sent to a part or one of its content elements. Semantic events pertain directly to the part's content model and can have meaning independent of the part's display context. For example, semantic events could direct a part to get, set, or delete data. Contrast with *user event* . See also *Open Scripting Architecture* .

semantic-event handler -A routine that executes in response to receiving a specific semantic event.

semantic interface -A set of OpenDoc objects that provides an interface to allow parts to receive messages (semantic events) from other parts, in the same document or in other documents.

semantics -The relationships between symbols and their meanings.

semaphore -An object used by applications for signalling purposes and for controlling access to serially reusable resources.

separator -In SAA Advanced Common User Access architecture, a line or color boundary that provides a visual distinction between two adjacent areas.

sequence number -A number that defines the position of a frame in its *frame group* .

serial dialog box -See *modal dialog box* .

serialization -The consecutive ordering of items.

serialize -To ensure that one or more events occur in a specified sequence.

serially reusable resource (SRR) -A logical resource or object that can be accessed by only one task at a time.

server application -An application that responds to OSA events requesting a service or information sent by client applications or scripting components (for example, by printing a list of files, checking the spelling of a list of words, or performing a numeric calculation). OSA event servers and clients can reside on the same local computer.

service -An OpenDoc component that, unlike a part editor, is not primarily concerned with editing and displaying parts. Instead, it provides a service to parts or documents, using the OpenDoc extension mechanism. Spelling checkers or database-access tools, for example, can be implemented as services.

session -(1) A routing mechanism for user interaction via the console; a complete environment that determines how an application runs and how users interact with the application. OS/2 can manage more than one session at a time, and more than one process can run in a session. Each session has its own set of environment variables that determine where OS/2 looks for dynamic-link libraries and other important files. (2) In the OS/2 operating system, one instance of a started program or command prompt. Each session is separate from all other sessions that might be running on the computer. The operating system is responsible for coordinating the resources that each session uses, such as computer memory, allocation of processor time, and windows on the screen. (3) A logical connection between two entities (such as an OS/2 program and a database server) that facilitates the transmission of information between the two entities. An application has the option to accept or reject a session request. Authentication of the requesting user may be required before a session can commence. See also *authentication* , *message block* , *port* .

session ID -A number that uniquely identifies a session.

Settings dialog box -A dialog box, accessible through the Part Info dialog box, that displays part-specific, custom Info properties.

settings extension -An OpenDoc extension class that you can use to implement a Properties notebook.

shadow -An object that refers to another object. A shadow is not a copy of another object, but is another representation of the object.

shadow box -The area on the screen that follows mouse movements and shows what shape the window will take if the mouse button is released.

shape -A description of a geometric area of a drawing canvas.

shared data -Data that is used by two or more programs.

shared memory -In the OS/2 operating system, a segment that can be used by more than one program.

shared resource -A facility used by multiple parts. Examples of shared resources are the menu focus, selection focus, keystroke focus, and serial ports. See also *arbitrator* .

shear -In computer graphics, the forward or backward slant of a graphics symbol or string of such symbols relative to a line perpendicular to the baseline of the symbol.

shell -(1) A software interface between a user and the operating system of a computer. Shell programs interpret commands and user interactions on devices such as keyboards, pointing devices, and touch-sensitive screens, and communicate them to the operating system. (2) Software that allows a kernel program to run under different operating-system environments.

shell plug-in -A shared library that modifies or extends the functions of the document shell.

shutdown -The process of ending operation of a system or a subsystem, following a defined procedure.

sibling -A frame or facet at the same level of embedding as another frame or facet within the same containing frame or facet. Sibling frames and facets are *z-ordered* to allow for overlapping.

sibling processes -Child processes that have the same parent process.

sibling windows -Child windows that have the same parent window.

signature -The aspect of a method defined by its return type and parameter list.

simple list -A list of like values; for example, a list of user names. Contrast with *mixed list* .

simple part -A part that cannot itself contain embedded parts. Contrast *container part* .

single-byte character set (SBCS) -A character set in which each character is represented by a one-byte code. Contrast with *double-byte character set* .

slider box -In SAA Advanced Common User Access architecture: a part of the scroll bar that shows the position and size of the visible information in a window relative to the total amount of information available. Also known as *thumb mark* .

small icon view type -A view type in which a part is represented by a 16 by 16-pixel bitmap image. Other possible view types for displaying a part include large icon, thumbnail, and frame.

SOM -See *System Object Model* .

SOM object -An object or class created according to the system object model.

source application -The application that sends a particular OSA event to another application or to itself. Typically, an OSA event client sends an OSA event requesting a service from an OSA event server; in this case, the client is the source application for the OSA event. The OSA event server may return a different OSA event as a reply; in this case, the server is the source for the reply OSA event.

source content -The content at the source of a link. It is copied into the link and then into the *destination content* .

source data -Statements in a scripting language that constitute an uncompiled script.

source file -A file that contains source statements for items such as high-level language programs and data description specifications.

source frame -(1) An embedded frame whose part that has been opened up into its own *part window* . (2) The frame to which other *synchronized frames* are attached.

source part -(1) In data transfer, the part that provides the data that is transferred. (2) For a link, the part that contains the original information that is copied and displayed at the destination of the Contrast with *destination part* .

source statement -A statement written in a programming language.

special handler dispatch table -A table in shared memory that the OSA Event Manager uses to keep track of various specialized handlers.

specific dynamic-link module -A dynamic-link module created for the exclusive use of an application.

spin button -In SAA Advanced Common User Access architecture, a type of entry field that shows a scrollable ring of choices from which a user can select a choice. After the last choice is displayed, the first choice is displayed again. A user can also type a choice from the scrollable ring into the entry field without interacting with the spin button.

spline -A sequence of one or more Bézier curves.

split-frame view -A display technique for windows or frames, in which two or more facets of a frame display different scrolled portions of a part's content.

spooler -A program that intercepts the data going to printer devices and writes it to disk. The data is printed or plotted when it is complete and the required device is available. The spooler prevents output from different sources from being intermixed.

stack -A list constructed and maintained so that the next data element to be retrieved is the most recently stored. This method is characterized as last-in-first-out (LIFO).

standard window -A collection of window elements that form a panel. The standard window can include one or more of the following window elements: sizing borders, system menu icon, title bar, maximize/minimize/restore icons, action bar and pull-downs, scroll bars, and client area.

static canvas -A drawing canvas that cannot be changed after it has been rendered, such as a printer page. Contrast with *dynamic canvas* .

static control -The means by which the application presents descriptive information (for example, headings and descriptors) to the user. The user cannot change this information.

static storage -(1) A read/write storage unit in which data is retained in the absence of control signals. (A) Static storage may use dynamic addressing or sensing circuits. (2) Storage other than *dynamic storage* . (A)

stationery -A part that opens by copying itself and opening the copy into a window, leaving the original stationery part unchanged.

storage system -The OpenDoc mechanism for providing persistent storage for documents and parts. The storage system object must provide unique identifiers for parts as well as cross-document links. It stores parts as a set of standard properties plus type-specific content data.

storage unit -In the OpenDoc storage subsystem, an object that represents the basic unit of persistent storage. Each storage unit has a list of properties, and each property contains one or more data streams called values.

storage-unit cursor -A preset storage unit/ property/value designation, created to allow swift focusing on frequently accessed data.

storage-unit ID -A unique identifier of a storage unit within a draft.

storage-unit view -A storage unit prefocused on a given property and value. A storage-unit view provides thread-safe access to a storage unit.

strong persistent reference -A persistent reference that, when the storage unit containing the reference is cloned, causes the referenced storage unit to be copied also. Contrast with *weak persistent reference* .

style -See *window style* .

subclass -An object class that inherits properties, element classes, and OSA events from another object class-its superclass. A subclass can also include properties, element classes, or OSA events that are not inherited from its superclass. Every object class, with the exception of cObject, is a subclass of another object class. See also *object class* , and *superclass* .

subdirectory -In an IBM personal computer, a file referred to in a root directory that contains the names of other files stored on the diskette or fixed disk.

subframe -A frame that is both an embedded frame in, and a display frame of, a part. A part can create an embedded frame, make it a subframe of its own frame, and then display itself in that subframe.

subsystem -A broad subdivision of the interface and capabilities of OpenDoc, involving one or more protocols (for example, OpenDoc subsystems include shell, storage, drawing, user events, and semantic events).

suite -In the *OSA Event Registry: Standard Suites* , a group of definitions for OSA events, object classes, primitive object classes, descriptor types, and constants that are all used for a set of related activities. For example, the Text suite includes definitions of OSA events, object classes, and so on that are used for text processing.

superclass -A class from which another class (its *subclass*) is derived. Also called ancestor, base class, or parent class. It is the object class from which a subclass inherits properties, elements, and OSA events. See also *inheritance* , *object class* , *subclass* .

swapping -(1) A process that interchanges the contents of an area of real storage with the contents of an area in auxiliary storage. (I) (A) (2) In a system with virtual storage, a paging technique that writes the active pages of a job to auxiliary storage and reads pages of another job from auxiliary storage into real storage. (3) The process of temporarily removing an active job from main storage, saving it on disk, and processing another job in the area of main storage formerly occupied by the first job.

switch -(1) In SAA usage, to move the cursor from one point of interest to another; for example, to move from one screen or window to another or from a place within a displayed image to another place on the same displayed image. (2) In a computer program, a conditional instruction and an indicator to be interrogated by that instruction. (3) A device or programming technique for making a selection, for example, a toggle, a conditional jump.

switch list -See *Task List* .

symbolic identifier -A text string that equates to an integer value in an include file, which is used to identify a programming object.

symbols -In Information Presentation Facility, a document element used to produce characters that cannot be entered from the keyboard.

synchronized frames -Separate frames that display the same representation of the same part, and should therefore be updated together. In general, if an embedded part has two or more editable display frames of the same presentation, those frames (and all their embedded frames) should be synchronized.

synchronous -Pertaining to two or more processes that depend upon the occurrence of specific events such as common timing signals. (T) See also *asynchronous* .

synthetic command ID -A command ID created by OpenDoc for a menu command that had not previously been registered with the menu bar object.

system coercion dispatch table -See *coercion handler dispatch table* .

System Menu -In the Presentation Manager, the pull-down in the top left corner of a window that allows it to be moved and sized with the keyboard.

system object accessor dispatch table -See *object accessor dispatch table* .

System Object Model (SOM) -A mechanism for language-neutral, object-oriented programming.

system OSA event dispatch table -See *OSA event dispatch table* .

system queue -The master queue for all pointer device or keyboard events.

system result handler -A result handler that is available to all applications that use the system. Contrast with *application result handler* .

system-defined messages -Messages that control the operations of applications and provides input an other information for applications to process.

Systems Application Architecture (SAA) -A set of IBM software interfaces, conventions, and protocols that provide a framework for designing and developing applications that are consistent across systems.

Glossary - T

table tags -In Information Presentation Facility, a document element that formats text in an arrangement of rows and columns.

tag -(1) One or more characters attached to a set of data that contain information about the set, including its identification. (I) (A) (2) In Generalized Markup Language markup, a name for a type of document or document element that is entered in the source document to identify it.

target address -An application signature, a process serial number, a session ID, a target ID record, or some other application-defined type that identifies the target of an OSA event.

target application -The application addressed to receive an OSA event. Typically, an OSA event client sends an OSA event requesting a service from a server application; in this case, the server is the target application of the OSA event. The server application may return a different OSA event as a reply; in this case, the client is the target of the reply OSA event.

target object -An object to which the user is transferring information.

Task List -In the Presentation Manager, the list of programs that are active. The list can be used to switch to a program and to stop programs.

terminate-and-stay-resident (TSR) -Pertaining to an application that modifies an operating system interrupt vector to point to its own location (known as hooking an interrupt).

terminology resource -A resource (of type 'aete') that is required for scriptability.

text -Characters or symbols.

text cursor -A symbol displayed in an entry field that indicates where typed input will appear.

text window -Also known as the VIO window.

text-windowed application -The environment in which the operating system performs advanced-video input and output operations.

thread -A unit of execution within a process. It uses the resources of the process.

thread-safe -Said of an activity, or access to data, that can be safely undertaken in a multitasking environment.

thumb mark -The portion of the scroll bar that describes the range and properties of the data that is currently visible in a window. Also known as a *slider box* .

thumbnail view type -A view type in which a part is represented by a large (64-by-64 pixels) bitmap image that is typically a miniature representation of the layout of the part content. Other possible view types for displaying a part include large icon, small icon, and frame.

thunk -Term used to describe the process of address conversion, stack and structure realignment, etc., necessary when passing control between 16-bit and 32-bit modules.

tilde -A mark used to denote the character that is to be used as a mnemonic when selecting text items within a menu.

time-critical process -A process that must be performed within a specified time after an event has occurred.

time slice -(1) An interval of time on the processing unit allocated for use in performing a task. After the interval has expired, processing-unit time is allocated to another task, so a task cannot monopolize processing-unit time beyond a fixed limit. (2) In systems with time sharing, a segment of time allocated to a terminal job.

timer -A facility provided under the Presentation Manager, whereby Presentation Manager will dispatch a message of class WM_TIMER to a particular window at specified intervals. This capability may be used by an application to perform a specific processing task at predetermined intervals, without the necessity for the application to explicitly keep track of the passage of time.

timer tick -See *clock tick* .

tinkerable -A level of scripting support of a part. A tinkerable part allows the user to customize it, changing its behavior during virtually any user action. Contrast with *scriptable* and *recordable* .

title bar -In SAA Advanced Common User Access architecture, the area at the top of each window that contains the window title and system menu icon. When appropriate, it also contains the minimize, maximize, and restore icons. Contrast with *panel title* .

TLB -Translation lookaside buffer.

token -A short, codified representation of a string. The session object creates tokens for ISO strings. In OSA events for OpenDoc, a special descriptor structure that a part uses to identify one or more content objects within itself.

token disposal function -An object callback function that disposes of a token.

transaction -(1) An exchange between a workstation and another device that accomplishes a particular action or result. (2) In OpenDoc, a sequence of OSA events sent back and forth between the client and server applications, beginning with the client's initial request for a service. All OSA events that are part of one transaction must have the same transaction ID.

transform -(1) The action of modifying a picture by scaling, shearing, reflecting, rotating, or translating. (2) The object that performs or defines such a modification; also referred to as a *transformation* .

translation -The conversion of one type of data to another type of data. Specifically, the conversion of data of one part kind to data of another part kind. Note that translation can involve loss of *fidelity* .

Translation lookaside buffer (TLB) -A hardware-based address caching mechanism for paging information.

Tree -In the Presentation Manager, the window in the *File Manager* that shows the organization of drives and directories.

truncate -(1) To terminate a computational process in accordance with some rule (A) (2) To remove the beginning or ending elements of a string. (3) To drop data that cannot be printed or displayed in the line width specified or available. (4) To shorten a field or statement to a specified length.

TSR -Terminate-and-stay-resident.

Glossary - U

undo -To rescind a command, negating its results. OpenDoc provides the ability to undo events by utilizing a command history.

unnamed pipe -A circular buffer, created in memory, used by related processes to communicate with one another. Contrast with *named* pipe.

unordered list -In Information Presentation Facility, a vertical arrangement of items in a list, with each item in the list preceded by a special character or bullet.

update region -A system-provided area of dynamic storage containing one or more (not necessarily contiguous) rectangular areas of a window that are visually invalid or incorrect, and therefore are in need of repainting.

update ID -(1) In OpenDoc, a number used to identify a particular instance of Clipboard contents. (2) A number used to identify a particular instance of link source data.

used shape -A shape that describes the portion of a frame that a part actually uses for drawing; that is, the part of the frame that the containing part should not draw over.

user event -A message, sent to a part by the dispatcher, that pertains only to the state of the part's graphical user interface, not directly to its contents. User events include mouse clicks and keystrokes, and they deliver information about, among other things, window locations and scroll bar positions. Contrast with s*emantic event* .

user interface -Hardware, software, or both that allows a user to interact with and perform operations on a system, program, or device.

user-interface part -A part without content elements, representing a unit of a document's user interface. Buttons and dialog boxes, for example, can be user-interface parts.

user property -One of a set of user-accessible characteristics of a part or its frame. The user can modify some user properties, such as the name of a part; the user cannot modify some other user properties, such as part category. Each user property defined by OpenDoc is stored as a distinct property in the storage unit of the part or its frame.

utility program -(1) A computer program in general support of computer processes; for example, a diagnostic program, a trace program, a sort program. (T) (2) A program designed to perform an everyday task such as copying data from one storage device to another. (A)

Glossary - V

validate -To mark a portion of a canvas (or facet, or frame) as no longer in need of redrawing. Contrast with *invalidate* .

value -In the OpenDoc storage subsystem, a data stream associated with a property in a storage unit. Each property has a set of values, and there can be only one value of a given data type for each property.

value set control -A visual component that enables a user to select one choice from a group of mutually exclusive choices.

vector font -A set of symbols, each of which is created as a series of lines and curves. Synonymous with *outline font* . Contrast with *image font* .

VGA -Video graphics array.

view -A way of looking at an object's information.

view type -The basic visual representation of a part. Supported view types include frame, icon, small icon, and thumbnail.

viewer -See *part viewer* .

viewing pipeline -The series of transformations applied to a graphic object to map the object to the device on which it is to be presented.

viewing window -A clipping boundary that defines the visible part of model space.

VIO -Video Input/Output.

virtual memory (VM) -Synonymous with *virtual storage* .

virtual storage -(1) The storage space that may be regarded as addressable main storage by the user of a computer system in which virtual addresses are mapped into real addresses. The size of virtual storage is limited by the addressing scheme of the computer system and by the amount of auxiliary storage available, not by the actual number of main storage locations. (I) (A) (2) Addressable space that is apparent to the user as the processor storage space, from which the instructions and the data are mapped into the processor storage locations. (3) Synonymous with *virtual memory* .

visible region -A window's presentation space, clipped to the boundary of the window and the boundaries of any overlying window.

volume -(1) A file-system driver that uses a block device driver for input and output operations to a local or remote device. (I) (2) A portion of data, together with its data carrier, that can be handled conveniently as a unit.

Glossary - W

weak persistent reference -A persistent reference that, when the storage unit containing the reference is cloned, is ignored; the referenced storage unit is not copied. Contrast with *strong persistent reference* .

whose descriptor record -A coerced AE record of descriptor type typeWhoseDescriptor. The OSA Event Manager creates whose descriptor records when it resolves object specifier records that specify formTest.

whose range descriptor record -A coerced AE record of type typeWhoseRange. Under certain conditions, the OSA Event Manager coerces a range descriptor record to a whose range descriptor record when it resolves object specifier records that specify formTest.

wildcard character -Synonymous with *global file-name character* .

window -(1) A portion of a display surface in which display images pertaining to a particular application can be presented. Different applications can be displayed simultaneously in different windows. (A) (2) An area of the screen with visible boundaries within which information is displayed. A window can be smaller than or the same size as the screen. Windows can appear to overlap on the screen. (3) A division of a screen in which one of several programs being executed concurrently can display information.

window canvas -The canvas attached to the root facet of a window. Every window has a window canvas.

window class -The grouping of windows whose processing needs conform to the services provided by one window procedure.

window-content transform -The composite transform that converts from a part's content coordinates to its window coordinates.

window coordinate space -The coordinate space of the window in which a part's content is drawn. It may or may not be equal to the canvas coordinate space.

window coordinates -A set of coordinates by which a window position or size is defined; measured in device units, or *pels* .

window-frame transform -The composite transform that converts from a part's frame coordinates to its window coordinates.

window handle -Unique identifier of a window, generated by Presentation Manager when the window is created, and used by applications to direct messages to the window.

window procedure -Code that is activated in response to a message. The procedure controls the appearance and behavior of its associated windows.

window rectangle -The means by which the size and position of a window is described in relation to the desktop window.

window resource -A read-only data segment stored in the .EXE file of an application o the .DLL file of a dynamic link library.

window state -An object that lists the set of windows that are open at a given time. Part editors can alter the window state, and the window state can be persistently stored.

window style -The set of properties that influence how events related to a particular window will be processed.

window title -In SAA Advanced Common User Access architecture, the area in the title bar that contains the name of the application and the OS/2 operating system file name, if applicable.

Workplace Shell -The OS/2 object-oriented, graphical user interface.

workstation -(1) A display screen together with attachments such as a keyboard, a local copy device, or a tablet. (2) One topor more programmable or nonprogrammable devices that allow a user to do work.

world-coordinate space -Coordinate space in which graphics are defined before transformations are applied.

world coordinates -A device-independent Cartesian coordinate system used by the application program for specifying graphical input and output. (I) (A)

wrapper -An object (or class) that exists to provide an object-oriented interface to a non-object- oriented or system-specific structure. The OpenDoc class ODWindow, for example, is a wrapper for a system-specific window structure.

WYSIWYG -What-You-See-Is-What-You-Get. A capability of a text editor to continually display pages exactly as they will be printed.

Glossary - X

There are no glossary terms for this starting letter.

Glossary - Y

There are no glossary terms for this starting letter.

Glossary - Z

z-order -The order in which sibling windows are presented. The topmost sibling window obscures any portion of the siblings that it overlaps; the same effect occurs down through the order of lower sibling windows.

z-ordering -The front-to-back ordering of sibling frames used to determine clipping and event handling when frames overlap.

zooming -The progressive scaling of an entire display image in order to give the visual impression of movement of all or part of a display group toward or away from an observer. (I) (A)
